SISTEMAS INFORMÁTICOS
Ingeniería en informática

Curso 2006-2007

Comprobación automática de la complejidad asintótica de programas Java

Esther Solanas González, Cristina Rivero Ortiz y María Concepción Moraleda de Haro

Directoras:
Purificación Arenas Sánchez
Elvira María Albert Albiol

UNIVERSIDAD COMPLUTENSE
MADRID
Facultad de Informática

Los abajo firmantes, Mª Concepción Moraleda de Haro, Cristina Rivero Ortiz y Esther Solanas González autoras del proyecto Comprobación automática de la complejidad asintótica de programas Java en la asignatura de Sistemas Informáticos, autorizan a la Universidad Complutense de Madrid a difundir y a utilizar con fines exclusivamente académicos, nunca comerciales, y mencionando expresamente a sus autores, los contenidos de este documento, así como el código, prototipos o documentación asociada a dicho proyecto

Esther Solanas González	Cristina Rivero Ortiz	Concepción Moraleda de Haro
Fdo:.....	Fdo:.....	Fdo:.....

Agradecimientos

Nos gustaría agradecer el apoyo y el cariño que nos demuestra cada día nuestras familias, gracias a las cuales ha sido posible no sólo que terminemos este proyecto sino que hayamos podido realizar un largo camino hasta llegar aquí.

También agradecer a nuestros novios la paciencia y el apoyo incondicional durante estos últimos meses de nervios y agobios.

No sé nos puede olvidar la ayuda entregada por nuestras directoras de proyecto que no han dejado de atendernos durante el transcurso del proyecto. Y también a algún miembro del CLIP que nos ha ayudado a solventar alguna duda.

Resumen

0.1. Resumen

El objetivo de nuestro proyecto consiste en comprobar automáticamente la complejidad asintótica de programas Java, utilizando como lenguaje para la implementación Prolog.
El proceso de análisis para realizar el chequeo de la complejidad está realizado sobre la información obtenida del bytecode resultante de la compilación del programa Java. Disponemos también de interfaces Web, para mostrar los resultados experimentales obtenidos, implementadas haciendo uso de tecnologías como Jsp, Servlets, CGI y de la librería Pillow (librería Prolog de dominio público).

0.2. Abstract

The objective of our project consists of verifying automatically the upper bounds of Java programs, using for the implementation the language Prolog. The process of analysis to make the ckecking of the complexity is made by means of the data obtained from bytecode resulting of the compilation of the Java program. We also have interfaces Web to show the obtained experimental results, implemented making use of technologies like Jsp, Servlets, cgi and of the Pillow bookstore (Prolog bookstore of public dominion).

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

La idea general del proyecto es la siguiente: dado un programa Java y una cierta complejidad (lineal, exponencial, etc, ...) lo que se pretende es comprobar si el programa tiene una complejidad igual o menor que la dada, para ello hemos desarrollado un prototipo en PROLOG, concretamente en CIAO PROLOG [13], que lleva a cabo la comprobación automática de las complejidades.
CIAO Prolog es un entorno de programación lógica y de restricciones GNU LGPL. Está desarrollado por el laboratorio de Computación Lógica, Implementación y Paralelismo (CLIP) de la Facultad de Informática de la Universidad Politécnica de Madrid (UPM).

Para realizar nuestra implementación , nos hemos servido de la documentación aportada por nuestras directoras (del departamento de sistemas informáticos y computación (DSIC) de la UCM) y sus colaboradores (del CLIP de la UPM). Toda la información utilizada fundamentalmente se encuentra en los siguientes artículos [7], [8], [9], [10] y [6]. Así mismo la implementación de nuestro sistema también se fundamenta en el código de su proyecto.

Hacemos un resumen del contenido de cada uno de los capítulos de los que consta el trabajo:

Capítulo 2. El bytecode de Java

Este capítulo consiste en una explicación minuciosa del bytecode de Java a través de un ejemplo que se explica paso a paso, así como una amplia descripción de la máquina virtual donde se ejecuta este bytecode.
El byecode es el fundamento de nuestro proyecto pues todo el análisis del

coste se realiza a partir del bytecode, podríamos decir que es la entrada a la aplicación.
Además tener un conocimiento profundo del bytecode de Java hace que los programadores sean mejores. Así como un compilador de C o C++ transforma código fuente en código ensamblador, los compiladores de Java transforman código Java en bytecode. Los programadores de Java deberían entender qué es el bytecode y cómo funciona y lo más importante, qué bytecode se genera a través de un compilador de Java. En muchos casos, el bytecode que se genera no es lo que uno esperaba. Para terminar, se puede decir que el bytecode tiene un papel fundamental en el tamaño y en la velocidad de ejecución del código.
La razón por la que el bytecode es la entrada al análisis del coste es que en muchas situaciones (en el contexto de código móvil, por ejemplo, donde los recursos son muy limitados) es improbable tener acceso al código fuente, esto es, sólo podemos tratar con código compilado. Es por esto que el bytecode de Java se está convirtiendo en uno de los formatos más populares y tener analizadores del coste para bytecode es por lo tanto muy deseable.

Capítulo 3. Generación de ecuaciones de coste

El análisis del coste del bytecode de Java genera en tiempo de compilación unas relaciones de coste que definen el coste de los programas como función del tamaño de sus datos de entrada. El objetivo del análisis del coste del bytecode es la generación de las ecuaciones de coste. La generación de las relaciones de coste atraviesa las siguientes etapas que explicamos con detalle a través de un ejemplo. Las etapas son las siguientes:

- Generación del grafo de control de flujo: Constituye la primera fase del análisis del coste. El bytecode asociado a un método se transforma en un grafo de control de flujo. Transformar el desestructurado flujo de control del bytecode en recursión es una parte fundamental de análisis. A grandes rasgos, dado un método m, se denota como G_m su CFG (Control Flow Graph) que es un grafo dirigido donde cada nodo representa un bloque. Cada bloque $Block_{id}$ es una tupla de la forma <id, G, B, D> donde: id es el identificador del bloque; G es el guarda del bloque que indica las condiciones bajo las que el bloque se ejecuta; B es una secuencia de instrucciones de bytecode contiguas para las cuales se garantiza que serán ejecutadas sin ningún tipo de condición (por ejemplo, si G tiene éxito entonces todas las instrucciones en B serán

ejecutadas antes de que el control se diriga a otro bloque); y D es la lista de adyacencia para el $Block_{id}$, por ejemplo si id' pertenece al conjunto D entonces hay un arco de $Block_{id}$ a $Block_{id'}$. Los guardas tienen la forma de guard(C), donde C es una condición booleana sobre las variables locales y los elementos de la pila.
Una gran parte de las instrucciones de bytecode tienen un único sucesor. Sin embargo, existen tres tipos de bifurcaciones:

- Saltos condicionales.
- Enlace dinámico.
- Excepciones.

A continuación mostramos el grafo de control de flujo para el bytecode de la multiplicación de matrices:

- Generación de la representación recursiva intermedia: Es la segunda fase del análisis. A partir del CFG se obtiene una representación recursiva del método en la que cada iteración se transforma en recursión y en la que la pila de operandos de la máquina virtual de Java 'se aplana' en el sentido de que su contenido se representa como una serie de variables locales.
  En esta representación cada bloque del grafo se representa como una regla.

- Generación de las relaciones de tamaño: La generación de las relaciones de tamaño constituye la tercera etapa de la generación de las relaciones de coste. Esta etapa consiste en inferir relaciones de tamaño entre variables de entrada que están en la cabeza de una regla y aquellas que están en una llamada a un bloque o en una llamada a un método dentro del cuerpo de la regla. Inferir relaciones de tamaño es esencial para definir el coste de un bloque en términos del coste de sus sucesores.
  Este tema se trata en detalle en este capítulo además de dar una completa definición de la noción de relación de tamaño.

- Cálculo de las variables relevantes: Constituye la cuarta etapa. Para cada regla de la representación recursiva se calcula el conjunto de argumentos de entrada que son importantes para hallar el coste del programa, es decir, se eliminan todas aquellas variables que no dependen de los guardas.
  En este capítulo se explica en detalle qué metodo se utiliza para hallar este conjunto de argumentos.
  que en la implementación se almacena a través del hecho Prolog llamado jcost(A, B, C, D, E). A partir de la representación recursiva, del conjunto de argumentos que se consideran importantes para el coste y de las relaciones de tamaño, en este paso se generan automáticamente las relaciones de coste. Como se ha dicho anteriormente, estas relaciones de coste expresan el coste de un método en función de sus argumentos de entrada y con la información contenida en ellos vamos a chequear la complejidad de los programas Java. De todas formas, es importante tener en cuenta que hay que realizar diversas transformaciones a las relaciones de coste que produce esta etapa para poder chequear la complejidad de un método. Estas técnicas solucionan los problemas en la representación y se explican detalladamente en un capítulo posterior.

Capítulo 4. Comprobación de complejidades

Este capítulo recopila todo lo que es necesario para realizar de manera satisfactoria el chequeo de la complejidad de un programa implementado en Java. Primeramente en la sección 'Esquemas utilizados' se detallan las ecuaciones de recurrencia. Estas ecuaciones de recurrencia indican como tiene que ser la forma de un método para que tenga una determina complejidad. Con la ayuda de estas ecuaciones y con la información contenida en las relaciones de coste hemos construido nuestros predicados Prolog que comprueban la complejidad de un método de una clase Java.
Así mismo, en este capítulo se explican los problemas que surgen de esta representación de las relaciones de coste y las soluciones a los mismos.
Los problemas son:

- La forma de representar los CFG y su correspondiente representación recursiva. Los CFG presentados tienen diversas deficiencias a la hora de utilizarlos en un análisis para inferir propiedades globales de un programa. Se proponen unos CFG que contienen diferentes subgrafos independientes para cada bucle del programa en vez de tener un único

CFG en el que los bucles están conectados. La ventaja está en que utilizando este tipo de CFG se produce una representación recursiva intermedia en la que los bucles se mantienen separados. Esta técnica se conoce como abstracción de bucles y es importante tener en cuenta que nuestro comprobador de complejidades no funciona a menos que se realice.

- No tener recursión directa en las relaciones de coste. Incluso para los bucles más simples dentro de un programa, su correspondiente relación de coste a veces contiene recursiones que no son directas. Se propone utilizar la evaluación parcial (EP), que es una técnica conocida para la optimización de programas. Por lo tanto, el propósito es transformar las relaciones de coste originales en una forma canónica en la que todas las recursiones sean directas y esto se hace a través de la EP.

Capítulo 5. Descripción del sistema

Este capítulo trata la implementación del sistema de comprobación de complejidades y el uso del mismo.
La implementación se divide en dos grandes partes: la implementación de los predicados Prolog que nos permiten comprobar si un determinado método tiene una determinada complejidad y la implementación de la interfaz web de nuestro sistema.

1. Implementación de los predicados Prolog:
   Para esta parte de la implementación nos hemos basado , como se ha dicho antes, en las relaciones de coste que constituyen la salida del sistema que han implementado nuestros directoras de proyeto y sus colaboradores. A partir de la información de estas relaciones de coste podemos verificar si el método de entrada tiene la complejidad de entrada. El contenido de los jcost que son las relaciones de coste se detallan en este capítulo de la memoria, así como la implementación de nuestros predicados.

2. Implementación de la interfaz web del sistema:
   Para implementar la interfaz hemos utilizado las siguientes tecnologías: CGI, Java Servlets y Jsp. En el navegador, desde la correspondiente Web elaborada con JSP, el usuario enviará los datos que serán utilizados para dar respuesta a la petición del cliente. El servlet con la

información de la jsp, sabrá lo que tiene que hacer con los datos y redirigirá el control para que se ejecuten los predicados o clases java para generar la respuesta a la petición del cliente que finalmente mostrará la información resultante de todo el proceso a través de una Web.

El uso del sistema es un manual para utilizar el sistema.

Capítulo 2

El Bytecode de Java

Un programa escrito en Java, debe pasar por un proceso que se llama compilación. Si este término es generalmente usado para el mecanismo encargado de convertir código fuente a código nativo de la plataforma sobre la que pretende ejecutarse, en Java la compilación produce un código intermedio (el bytecode) destinado a ser ejecutado por la máquina virtual de Java. Este bytecode toma la forma de un fichero con extensión ".class". Todo programa en Java está compuesto por una o varias clases, y cada clase estará en un archivo diferente, tanto antes como después de la compilación.

Para facilitar la distribución de grandes aplicaciones, las clases que lo forman pueden ser empaquetadas juntas, tras su compilación, en un único archivo (con extensión ".jar"). Este archivo puede entonces ser ejecutado por la máquina virtual de Java.

La primera máquina virtual de Java (JVM o Java Virtual Machine), desarrollada por Sun MicroSystems, es una máquina virtual que ejecuta el código resultante de la compilación de un programa Java, el bytecode de Java. Aunque compiladores de otros lenguajes pueden dar lugar a bytecode ejecutable sobre la JVM.

Lo más importante:

- El compilador de Java genera un código intermedio independiente de la plataforma (bytecode).

- Los bytecodes pueden considerarse como el lenguaje máquina de una máquina virtual, la máquina virtual de Java.

- Cuando se quiere ejecutar una aplicación Java, al cargar el programa en memoria, se puede:

  - Interpretar los bytecodes instrucción por instrucción.

  - Compilar los bytecodes para obtener el código máquina necesario para ejecutar la aplicación en el ordenador (compilador JIT [Just In Time]).

De esta forma, se puede ejecutar un programa escrito en Java sobre distintos sistemas operativos (Windows, Solaris, Linux,...) sin tener que recompilarlo, como sucedería con programas escritos en lenguajes como C.

2.1. La máquina virtual de Java

2.1.1. Introducción

[20] La máquina virtual de Java (en inglés Java Virtual Machine, JVM) es un programa nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (como hemos dicho antes, el bytecode de Java), el cual es generado por el compilador del lenguaje Java.

La gran ventaja de la máquina virtual de Java es aportar portabilidad al lenguaje de manera que desde SUN se han creado diferentes máquinas virtuales de Java para diferentes arquitecturas y así un programa .class escrito en Windows puede ser interpretado en un entorno Linux. Tan solo es necesario disponer de dicha máquina virtual para dichos entornos.

La máquina virtual de Java puede estar implementada en software, hardware, una herramienta de desarrollo o un Web browser; lee y ejecuta código precompilado bytecode que es independiente de la plataforma multiplataforma. La JVM provee definiciones para un conjunto de instrucciones, un conjunto de registros, un formato para archivos de clases, la pila, un heap con recolector de basura y un área de memoria. Cualquier implementación de la JVM que sea aprobada por SUN debe ser capaz de ejecutar cualquier clase que

cumpla con la especificación.

Existen varias versiones, en orden cronológico, de la máquina virtual de Java. En general la definición del bytecode de Java no cambia significativamente entre versiones, y si lo hacen, los desarrolladores del lenguaje procuran que exista compatibilidad hacia atrás con los productos anteriores.

Por lo tanto, se puede concluir que: La JVM es una de las piezas fundamentales de la plataforma Java. Básicamente se sitúa en un nivel superior al Hardware del sistema sobre el que se pretende ejecutar la aplicación, y actúa como un puente que entiende tanto el bytecode, como el sistema sobre el que se pretende ejecutar. Así, cuando se escribe una aplicación Java, se hace pensando que será ejecutada en una máquina virtual de Java en concreto, siendo ésta la que, en última instancia, convierte de código bytecode a código nativo del dispositivo final.

Para poder ejecutar una aplicación en una Máquina Virtual de Java, el programa código debe compilarse de acuerdo a un formato binario portable estandarizado, normalmente en forma de ficheros con extensión .class. Un programa puede componerse de múltiples clases, en cuyo caso cada clase tendrá asociada su propio archivo .class. Para facilitar la distribución de aplicaciones, los archivos de clase pueden empaquetarse juntos en un archivo con formato jar. Esta idea apareció en la época de los primeros applets de Java. Estas aplicaciones pueden descargar aquellos archivos de clase que necesitan en tiempo de ejecución, lo que suponía una sobrecarga considerable para la red en una época donde la velocidad suponía un problema. El empaquetado evita la sobrecarga por la continua apertura y cierre de conexiones para cada uno de los fragmentos necesarios.

El código resultante de la compilación es ejecutado por la JVM que lleva a cabo la emulación del conjunto de instrucciones, bien por un proceso de interpretación o más habitualmente mediante un compilador JIT (Just In Time), como el HotSpot de Sun. Esta última opción convierte el bytecode a código nativo de la plataforma destino, lo que permite una ejecución mucho más rápida.

El inconveniente es el tiempo necesario al principio para la compilación.

En un sentido amplio, la Máquina Virtual de Java actúa como un puente entre el resultado de la compilación (el bytecode) y el sistema sobre el que se ejecuta la aplicación. Para cada dispositivo debe haber una JVM específica,

ya sea un teléfono móvil, un PC con Windows XP, o un microondas. En cualquier caso, cada máquina virtual conoce el conjunto de instrucciones de la plataforma destino, y traduce un código escrito en lenguaje Java (común para todas) al código nativo que es capaz de entender el Hardware de la plataforma.

2.1.2. El verificador del bytecode

La JVM verifica todo bytecode antes de ejecutarlo. Esto significa que solo una cantidad limitada de secuencias de bytecode forman programas válidos, por ejemplo una instrucción JUMP (branch) puede apuntar solo a una instrucción dentro de la misma función. A causa de esto, el hecho de que JVM es una arquitectura de pila no implica una carga en la velocidad para emulación sobre arquitecturas basadas en registros cuando usamos un compilador JIT: no hay diferencia para un compilador JIT si nombra registros con nombres imaginarios o posiciones de pila imaginarias que necesitan ser ubicadas a los registros de la arquitectura objetivo. De hecho, la verificación de código hace a la JVM diferente de una arquitectura clásica de pila cuya emulación eficiente con un compilador JIT es más complicada y típicamente realizado por un interprete más lento.

La verificación de código también asegura que los patrones de bits arbitrarios no pueden usarse como direcciones. La Protección de Memoria se consigue sin necesidad de una unidad de Gestión de Memoria(MMU). Así, JVM es una forma eficiente de obtener protección de memoria en chips que no tienen MMU.

[15]La máquina virtual de Java ejecuta programas bytecode que se pueden enviar desde lugares de la red que posiblemente no sean muy confiables. Esto se debe a que el código transmitido puede haber sido escrito maliciosamente o puede haberse corrompido durante su transmisión. La máquina vitual de Java contiene un verificador de bytecode para comprobar posibles errores del código antes de que este sea ejecutado, es decir el verificador de bytecode de JVM ejecuta una serie de chequeos sobre el bytecode antes de que este se ejecute. Este verificador es esencial para la seguridad del sistema, para citar un ejemplo específico, un virus en una versión nueva del verificador de bytecode de SUN permitió que los applets pudieran crear ciertos objetos del sistema los cuales en realidad no podían crear, como por ejemplo ClassLoaders[4]. El problema fue causado por un error en la forma de verificar las constructoras. En un estudio [14] se identificó otro error en el verificador de SUN que

permitía que código JVML utilizara un objeto que no había sido inicializado correctamente. Una cantidad importante de lagunas e inconsistencias han sido registradas en [[18], [5], [23], [3]]. Problemas como estos demuestran la necesidad de una especficación formal correcta del verificador del bytecode de Java, que se puede encontrar en [15].

2.1.3. Semántica operacional

El entorno de ejecución para programas JVML consiste en una pila de registros de activación y en un almacenamiento de objetos (en inglés heap). Los registros de activación se añaden a la pila de registros de activación en las invocaciones a métodos o cuando ocurre una excepción. Se eliminan de la pila cuando el método se ejecuta completamente o cuando se captura la excepción.

Cada registro de activación asociado a un método contiene:

- `El descriptor del método M.`
- `Un contador de programa pc`, que almacena la dirección de la siguiente instrucción de bytecode que se va a ejecutar.
- `Una pila de operandos s`, que almacena valores intermedios utilizados al ejecutar las instrucciones de bytecode.
- `Un conjunto de variables locales f` asociadas al método.
- `Información de inicialización z` para el objeto que se está inicializando dentro de una constructora.

Los registros de activación para las excepciones contienen sólo un puntero al objeto (dentro del heap) que ha sido lanzado debido a una excepción.

La JVM se representa como una máquina de estados, donde cada estado o configuración es de la forma:

$$C \equiv A;\ h$$

A representa la pila de registros de activación y h representa el heap. Cada paso de ejecución tiene la forma $\Gamma \vdash C_0 \rightarrow C_1$, cuyo pretendido significado es que a partir de la configuración C_0 en Γ se obtiene la configuración C_1 en un único paso.

Representación de la pila de registros de activación

La pila de registros de activación A es una sucesión de registros de activación. Una pila de registros de activación ha de tener una de las dos formas que se presentan a continuación:

- $\langle$M, pc, f, s, z$\rangle$ donde M es un descriptor del método, pc es el contador de programa, f es el conjunto de variables locales de M, s es la pila de operandos de M y z contiene información de inicialización de los objetos que se están inicializando dentro de una constructora.

- $\langle e \rangle_{exc}$ donde es un puntero al objeto que ha sido alcanzado por una excepción. En este caso, será siempre una referencia al objeto Throwable.

Representación de los objetos en el heap

El heap contiene todos los objetos y los arrays declarados dentro de un programa. Objetos de la clase σ son registros:

$$\{\!\{\langle \sigma_1, l_1, \tau_1 \rangle = v_1, \ldots, \langle \sigma_n, l_n, \tau_n \rangle = v_n \}\!\}_\sigma$$

Donde σ_i son nombres de clases, l_i nombres de atributos u τ_i son tipos de atributos (el mismo utilizado en el correspondiente descriptor de atributo dentro de la clase σ_i). La semántica estática asegurará que $\Gamma \vdash <: \sigma <: \sigma_{i1} <: \sigma_{i2} <: \ldots <: \sigma ik$, $i_p \neq i_m\ \forall$ p $\neq$ m y $\{i_1, \ldots, i_k\} = \{1, \ldots, n\}$ (no hay ciclos y herencia lineal).

Los arrays se almacenan en un registro del tipo:

$$[\![v_0, \ldots, v_n]\!]_{(\texttt{Array}\ \tau)}$$

El heap h es un mapa parcial a partir de lugares LOC hasta registros:

$$\begin{array}{lll} h : \text{LOC} \rightarrow & \langle\!\langle \langle \sigma_i, l_i \rangle = v_i \rangle\!\rangle_{\sigma}^{i \in I} & \text{(Object)} \\ & \langle\!\langle \langle \sigma_i, l_i \rangle = v_i \rangle\!\rangle_{\varphi\diamond(\texttt{Uninit } \sigma\ j)}^{i \in I} & \text{(uninitialized object)} \\ & [v_i]_{(\texttt{Array } \tau)}^{i \in \{0,\ldots n-1\}} & \text{(Array)} \end{array}$$

El Tag para un registro de un objeto que no está inicializado $\varphi\diamond(\texttt{Uninit } \sigma\ j)$ contiene dos informaciones. La primera parte φ indica que el objeto fue inicialmente creado con una instrucción new φ. La segunda parte indica que el programa creó el objeto en la línea pc del método y que debe llamar a una constructora de la clase σ del objeto antes de realizar cualquier operación con él. Por ejemplo, la ejecución de new σ genera el objeto $\sigma \diamond (Uninit\sigma pc)$. Ahora, si una constructora del tipo σ fuera llamada sobre este objeto, el tag se convertiría en $\sigma \diamond$ (Uninit σ 0). Si una constructora de σ', la super clase de σ, fuera a continuación invocada sobre el objeto, el tag se convertiría en $\varphi\diamond$ (Uninit σ' j), y así sucesivamente. Si se invoca la constructora de la clase 'Object' sobre el objeto su tag cambia a σ (el objeto se incializa completamente).

Tags

Todos los registros de la sección anterior tienen tags que dan soporte a los test del tiempo de ejecución. La función Tag recibe como entrada el heap h y un valor v y devuelve su tag asociado.

$$Tag(h, v) = \begin{cases} \texttt{int} & \text{if } v \text{ is an integer} \\ \texttt{float} & \text{if } v \text{ is a real} \\ \texttt{Null} & \text{if } v = \texttt{null} \\ \tau & \text{if } v \in \text{LOC and } h[v] = \langle\!\langle \ldots \rangle\!\rangle_{\tau} \text{ or } h[v] = [\ldots]_{\tau} \end{cases}$$

Como notación, $h[a].\{\!|\sigma, l, \tau|\!\}_F$ devuelve el valor del atributo $\{\!|sigma, l, \tau|\!\}_F$ a partir de la instancia en el lugar 'a' dentro del heap 'h', y se crea un nuevo heap con un valor modificado para ese atributo utilizando la notación $h[a.\{\!|\varphi, l, \tau|\!\}_F \mapsto v]$.

Instrucciones de bytecode

La JVM tiene instrucciones para los siguientes grupos de tareas:

- **Carga y Almacenamiento.**

- `arrayload`

$$\Gamma \vdash \langle M, pc, f, k.b.s, z\rangle.A; h \to \langle M, pc+1, f, v_k.s, z\rangle.A; h$$

donde:

  - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] =$ `arrayload` τ.
  - $h[b] = [\![v_0, \ldots, v_{n-1}]\!]_{(\texttt{Array } \tau')}$.
  - $\Gamma \vdash \tau' <: \tau$.
  - $0 \leq k \leq n$.

- `arraystore`

$$\Gamma \vdash \langle M, pc, f, v'.k.b.s, z\rangle.A; h \to \langle M, pc+1, f, s, z\rangle.A; h[b \mapsto [\![v_0, \ldots, v_{k-1}, v_k, v_{k+1}, \ldots, v_{n-1}]\!]_{(\texttt{Array } \tau')}]$$

donde:

  - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] =$ `arraystore` τ.
  - $h[b] = [\![v_0, \ldots, v_{n-1}]\!]_{(\texttt{Array } \tau')}$.
  - $\Gamma \vdash \tau' <: \tau$.
  - $\Gamma \vdash \mathit{Tag}(h, v') <: \tau'$.
  - $0 \leq k \leq n$.

- `store`

$$\Gamma \vdash \langle M, pc, f, v.s, z\rangle.A; h \to \langle M, pc+1, f[x \mapsto v], s\rangle.A; h$$

donde:

  - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] =$ `store` x.

- `load`

$$\Gamma \vdash \langle M, pc, f, s, z\rangle.A; h \to \langle M, pc+1, f, f[x].s\rangle.A; h$$

donde:

  - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] =$ `load` x.

- **Aritméticas.**

- `add`

$$\Gamma \vdash \langle M, pc, f, v_1.v_2.s, z\rangle.A; h \to \langle M, pc+1, f, (v_1 +_\tau v_2).s, z\rangle.A; h$$

donde:

- $\Gamma[M] = \langle P, H\rangle$ y $P[pc] = \texttt{add}$.
- $Tag(h, v_1) = Tag(h, v_2) = \tau$.

- **Conversión de tipos.**

- **Creación y manipulación de objetos.**

  - `getField`

    $\Gamma \vdash \langle M, pc, f, b.s, z\rangle.A; h \rightarrow \langle M, pc+1, f, h[b].\{\!|\varphi, l, \tau|\!\}_F.s, z\rangle.A; h$

    donde:

    - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] = \texttt{getfield}\ \{\!|\varphi, l, \tau|\!\}_F$.
    - $b \in Dom(h)$, por ejemplo, es una posición.
    - $\Gamma \vdash Tag(h, b) <: \varphi$ (b es un puntero a un objeto de una subclase de φ).

  - `putField`

    $\Gamma \vdash \langle M, pc, f, v.b.s, z\rangle.A; h \rightarrow \langle M, pc{+}1, f, s, z\rangle.A; h[b.\{\!|\varphi, label, \tau|\!\}_F \mapsto v]$

    donde:

    - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] = \texttt{putfield}\ \{\!|\varphi, label, \tau|\!\}_F$.
    - $\Gamma \vdash Tag(h, b) <: \varphi$.
    - $\Gamma \vdash Tag(h, v) <: \tau$.
    - $b \in Dom(h)$.

  - `new`
    $\Gamma \vdash \langle M, pc, f, s, z\rangle.A; h \rightarrow \langle M, pc+1, f, b.s, z\rangle.A; h[b \mapsto$
    $Blank(\sigma \diamond (\texttt{Uninit}\ \sigma\ pc))$ donde:

    - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] = \texttt{new}\ \sigma$.
    - $b \in Dom(h)$ (nueva posición).
    - La semántica utiliza la función *Blank* para crear nuevos registros. La función se define de forma separada para cada uno de los tres tipos de registros:
      - $Blank(\sigma) = \langle\!\langle \{\!|\sigma_i, l_i, \tau_i|\!\}_F = Zero(\tau_i)\rangle\!\rangle_{\sigma}^{i \in I}$.
      - $Blank(\sigma \diamond (\texttt{Uninit}\ \sigma'\ j)) = \langle\!\langle \{\!|\sigma_i, l_i, \tau_i|\!\}_F = Zero(\tau_i)\rangle\!\rangle_{\sigma \diamond (\texttt{Uninit}\ \sigma'\ j)}^{i \in I}$.
      - $Blank((\texttt{Array}\ \tau), n) = [v_i = Zero(\tau_i)]_{(\texttt{Array}\ \tau)}^{i \in \{0 \ldots n-1\}}$.

donde $\Gamma[\sigma].\texttt{fields} = \{\!|\sigma_i, \texttt{l}_i, \tau_i|\!\}_F\}^{i \in I}$. La función *Zero* calcula valores por defecto para cada tipo:

$$Zero(\tau) = \begin{cases} \texttt{0} & \text{if } \tau \text{ es un integer} \\ \texttt{0,0} & \text{if } \tau \text{ es un float} \\ \texttt{Null} & \text{if } \tau \in Ref \end{cases}$$

- `newarray`

  $\Gamma \vdash \langle M, pc, f, n.s, z\rangle.A; h \rightarrow \langle M, pc{+}1, f, b.s, z\rangle.A; h[b \mapsto Blank((\texttt{Array}\ \tau), n)]$

  donde:
  - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] = \texttt{newarray}\ \tau$.
  - $b \notin Dom(h)$ (nueva posición).

- **Gestión de pilas (push / pop).**

  - `push v`

    $$\Gamma \vdash \langle M, pc, f, s, z\rangle.A; h \rightarrow \langle M, pc+1, f, v.s, z\rangle.A; h$$

    donde:
    - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] = \texttt{push}\ v$.
  - `pop`

    $$\Gamma \vdash \langle M, pc, f, v.s, z\rangle.A; h \rightarrow \langle M, pc+1, f, s, z\rangle.A; h$$

    donde:
    - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] = \texttt{pop}$.

- **Transferencias de Control (branching).**

  - `ifeq L`

    $$\Gamma \vdash \langle M, pc, f, v_1.v_2.s, z\rangle.A; h \rightarrow \langle M, pc+1, f, s, z\rangle.A; h$$

    donde:
    - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] = \texttt{ifeq}\ L$.
    - $v_1 \neq v_2$.
  - `ifeq L`

    $$\Gamma \vdash \langle M, pc, f, v_1.v_2.s, z\rangle.A; h \rightarrow \langle M, L, f, s, z\rangle.A; h$$

    donde:

- $\Gamma[M] = \langle P, H\rangle$ y $P[pc] =$ `ifeq` L.
- $v_1 = v_2$.

- `goto L`

$$\Gamma \vdash \langle M, pc, f, s, z\rangle.A; h \rightarrow \langle M, L, f, s, z\rangle.A; h$$

donde:

- $\Gamma[M] = \langle P, H\rangle$ y $P[pc] =$ `goto` L.

- **Invocación y retorno a Métodos.**

- `jsr`

$$\Gamma \vdash \langle M, pc, f, s, z\rangle.A; h \rightarrow \\ \langle M, L, f, pc + 1.s, z\rangle.A; h$$

donde:

- $\Gamma[M] = \langle P, H\rangle$ y $P[pc] =$ `jsr` L.

- `ret`

$$\Gamma \vdash \langle M, pc, f, s, z\rangle.A; h \rightarrow \\ \langle M, f[x], f, s, z\rangle.A; h$$

donde:

- $\Gamma[M] = \langle P, H\rangle$ y $P[pc] =$ `ret` x.

- `invokespecial (1)`

$\Gamma \vdash \langle M, pc, f, s_1, \ldots, s_n.a.s, z\rangle.A; h \rightarrow \langle N, 1, f_0[0 \mapsto b, 1 \mapsto \_ \ldots n \mapsto \_], \epsilon, \langle b, \texttt{null}\rangle\rangle$.
$\langle M, pc, f, s_1 \ldots s_n.a.s, z\rangle.A; h[b \mapsto \mathit{Blank}(\varphi_0 \diamond (\texttt{Uninit}\ \varphi\ 0))]$ donde:

- $\Gamma[M] = \langle P, H\rangle$ and $P[pc] =$ `invokespecial` $\{\!|\varphi, <$ `init` $>, \alpha_1 \ldots \alpha_n \rightarrow$ `void`$|\!\}$.
- $\varphi \neq$ `Object`.
- $\mathit{Tag}(h, a) = \varphi_0 \diamond (\texttt{Uninit}\ \varphi'\ j)$.
- $b \notin \mathit{Dom}(h)$.
- $\varphi = \varphi' \wedge (\Gamma[\varphi'].\texttt{super} = \varphi \wedge \texttt{j} = 0)$.
- $N = \{\!|\varphi, <$ `init` $>, \alpha_1 \ldots \alpha_n \rightarrow$ `void`$|\!\}_M$.

- `invokespecial (2)`

$$\Gamma \vdash \langle M, pc, f, b.s, \langle b, \texttt{null} \rangle \rangle.A; h \rightarrow \\ \langle M, pc + 1, [c/b]f, [c/b]s, \langle b, c \rangle \rangle.A; h[c \mapsto \mathit{Blank}(\varphi))]$$

  donde:

  - $\Gamma[M] = \langle P, H \rangle$ and $P[pc] =$ `invokespecial N`
  - $N = \{\!|\texttt{Object}, < \texttt{init} >, \epsilon \rightarrow \texttt{void}|\!\}$.
  - $\mathit{Tag}(h, b) = \varphi \diamond (\texttt{Uninit}\ \sigma\ 0)$.
  - $c \notin \mathit{Dom}(h)$.
  - $\Gamma[\sigma].\texttt{super} = \texttt{Object}$.

- `invokespecial (3)`

$$\Gamma \vdash \langle M, pc, f, b.s, z \rangle.A; h \rightarrow \langle M, pc + 1, [c/b]f, [c/b]s, z \rangle.A; h[c \mapsto \mathit{Blank}(\texttt{Object}))]$$

  donde:

  - $\Gamma[M] = \langle P, H \rangle$ and $P[pc] =$ `invokespecial N`
  - $N = \{\!|\texttt{Object}, < \texttt{init} >, \epsilon \rightarrow \texttt{void}|\!\}$.
  - $\mathit{Tag}(h, b) = \texttt{Object} \diamond (\texttt{Uninit Object}\ j)$.
  - $c \notin \mathit{Dom}(h)$.
  - $j \neq 0$.

- `return (Constructoras 1)`

$$\Gamma \vdash \langle M, pc, f, s, \langle b, c \rangle \rangle.\langle M', pc' f', s_1 \ldots s_n, a.s', \langle a, \texttt{null} \rangle \rangle A; h \rightarrow \\ \langle M', pc' + 1, [c/a]f', [c/a]s', \langle a, c \rangle \rangle.A; h$$

  donde:

  - $\Gamma[M] = \langle P, H \rangle$ y $P[pc] =$ `return`
  - $M = \{\!|\sigma, < \texttt{init} >, \alpha_1, \ldots, \alpha_n \rightarrow \texttt{void}|\!\}$
  - $c \neq NULL$.

- `return (Constructoras 2)`

$$\Gamma \vdash \langle M, pc, f, s, \langle b, c \rangle \rangle.\langle M', pc' f', s_1 \ldots s_n, a.s', z' \rangle A; h \rightarrow \\ \langle M', pc' + 1, [c/a]f', [c/a]s', z' \rangle.A; h$$

  donde:

  - $\Gamma[M] = \langle P, H \rangle$ y $P[pc] =$ `return`
  - $M = \{\!|\sigma, < \texttt{init} >, \alpha_1, \ldots, \alpha_n \rightarrow \texttt{void}|\!\}$
  - $c \neq NULL$.

- $z' \neq \langle a, \texttt{null} \rangle$.

- `invokevirtual`

  $$\Gamma \vdash \langle M, pc, f, s_1 \dots s_n.b.s, z \rangle.A; h \rightarrow \langle N, 1, f_0[0 \mapsto b, 1 \mapsto \_ \dots n \mapsto \_], \epsilon, \emptyset \rangle.\langle M, pc, f, s_1 \dots s_n.b.s, z \rangle.A; h$$

  donde:

  - $\Gamma[M] = \langle P, H \rangle$ y $P[pc] = \texttt{invokevirtual}\ \{\!|\varphi, name, \alpha_1 \dots \alpha_n \rightarrow \gamma|\!\}$.
  - $Tag(h, b) = \sigma$.
  - $N = \{\!|\sigma, name, \alpha_1 \dots \alpha_n \rightarrow \gamma|\!\}_M$.
  - $\Gamma \vdash \sigma <: \varphi$.
  - "_"da soporte para los valores arbitrarios.
  - ¿Qué ocurre con estos tipos? ¿No hay control? El análisis estático asegura que el programa está bien tipado.

- `returnval (elimina un registro de activación).`

  $$\Gamma \vdash \langle M, pc, f, v.s, z \rangle.\langle M', pc', f', s_1 \dots s_n.b.s', z' \rangle.A; h \rightarrow \langle M', pc' + 1, f', v.s', z' \rangle.A; h$$

  donde:

  - $\Gamma[M] = \langle P, H \rangle$ y $P[pc] = \texttt{returnval}$.
  - $M = \{\!|\sigma, name, \alpha_1 \dots \alpha_n \rightarrow \gamma|\!\}_M$.
  - $\gamma \neq \texttt{void}$.

- `return (Métodos 1)`

  $$\Gamma \vdash \langle M, pc, f, s, z \rangle.\langle M', pc' f', s_1 \dots s_n, b.s', z' \rangle A; h \rightarrow \langle M', pc' + 1, f', s', z' \rangle.A; h$$

  donde:

  - $\Gamma[M] = \langle P, H \rangle$ y $P[pc] = \texttt{return}$
  - $M = \{\!|\sigma, m, \alpha_1, \dots, \alpha_n \rightarrow \texttt{void}|\!\}$
  - $m \neq <\texttt{init}>$.

- `return (Métodos 2)`

  $$\Gamma \vdash \langle M, pc, f, s, z \rangle.\epsilon; h \rightarrow \epsilon; h$$

  donde:

  - $\Gamma[M] = \langle P, H \rangle$ y $P[pc] = \texttt{return}$
  - $M = \{\!|\sigma, m, \epsilon \rightarrow \texttt{void}|\!\}$

- $m \neq <$ `init` $>$.

- **Lanzar excepciones.**

- `throw`
  Una excepción se puede generar debido a una declaración `throw` o un fallo en una comprobación en tiempo de ejecución.

$$\Gamma \vdash \langle M, pc, f, b.s, z\rangle.A; h \rightarrow \langle b\rangle_{exc}.\langle M, pc, f, b.s, z\rangle.A; h$$

  donde:

  - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] =$ `throw` σ.
  - $b \in Dom(h)$.
  - $\Gamma \vdash Tag(h, b) <:$ `Throwable`.

- `Tratamiento de una excepción (1)`

$$\Gamma \vdash \langle b\rangle_{exc}.\langle M, pc, f, s, z\rangle.A; h \rightarrow \langle b\rangle_{exc}.A; h$$

  donde:

  - $Tag(h, b) = \sigma$.
  - $CorrectHandler(\Gamma, M, pc, \sigma) = 0$.

  $CorrectHandler(\Gamma, M, pc, \sigma) = t$ if $\langle b, e, t, \sigma'\rangle$ es el primer handler (manejador) encontrado en la lista de handlers para M tal que $b \leq pc \leq e$ and $\Gamma \vdash \sigma <: \sigma'$. Si no se encuentra este handler entonces $CorrectHandler(\Gamma, M, pc, \sigma) = 0$.

- `Tratamiento de una excepción 2`

$$\Gamma \vdash \langle b\rangle_{exc}.\langle M, pc, f, s, z\rangle.A; h \rightarrow \langle M, t, f, b.\epsilon, z\rangle.A; h$$

  - $CorrectHandler(\Gamma, M, pc, \sigma) = t$.
  - $Tag(h, b) = \sigma$.

- `Tratamiento de una excepción 3`

$$\Gamma \vdash \langle b\rangle_{exc}.\epsilon; h \rightarrow \epsilon; h$$

- `arraystore (1)`

$$\Gamma \vdash \langle M, pc, f, v'.k.b.s, z\rangle.A; h \rightarrow \\ \langle e\rangle_{exc}.\langle M, pc, f, v'.k.b.s, z\rangle.A; h[e \mapsto Blank(Throwable)]$$

  donde:

  - $\Gamma[M] = \langle P, H\rangle$ y $P[pc] =$ `arraystore`.
  - $b = NULL$ and $e \notin Dom(h)$.

- `arraystore (2)`

$$\Gamma \vdash \langle M, pc, f, v'.k.b.s, z\rangle.A; h \rightarrow \\ \langle e\rangle_{exc}.\langle M, pc, f, v'.k.b.s, z\rangle.A; h[e \mapsto Blank(Throwable)]$$

  donde:

  - $\Gamma[M] = \langle P, H\rangle$ and $P[pc] =$ `arraystore`.
  - $H[b] = [v_0, \ldots, v_n]_{(\texttt{Array } \tau)}$.
  - $Tag(h, v') = \tau'$.
  - $e \notin Dom(h)$.
  - $\Gamma \not\vdash \tau <: \tau$.

La clave es la compatibilidad binaria. Cada sistema operativo de un host particular necesita su propia implementación de JVM y runtime. Estas JVMs interpretan el bytecode semánticamente de la misma manera, pero la implementación actual puede variar. Más complicado que solo la emulación de bytecode es la implementación compatible y eficiente de las APIs java las cuales tienen que ser mapeadas para cada sistema operativo de host.

2.1.4. Semántica estática

Entornos bien formados

Γ está bien formado si al menos verifica los siguientes requerimientos:

- No hay circularidades en la jerarquía de clases.

- Todas las clases que implementan sus interfaces declaradas por medio de definir todos los métodos que están en dichas interfaces.

- Una clase hereda todos los atributos y todas las interfaces declaradas de su superclase, y hereda o sobreescribe sus métodos.

Métodos válidos

(1) dice que el método no es una constructora. En (2), la función F_{TOP} asigna a todas las variables el valor Top. (3) inicializa la pila de tipos a la secuencia vacía. (4) demuestra que cualquier instrucción de bytecode en un método está bien tipada (deacuerdo con las reglas que se muestran más abajo). (5) asegura que todos los manejadores de excepciones para el método están también bien tipados.

$$(meth\ code)\ \frac{\begin{array}{cc} m \neq <\texttt{init}> & (1) \\ F_1 = F_{TOP}[0 \mapsto \sigma, 1 \mapsto \_, \ldots, n \mapsto \_] & (2) \\ S_1 = \epsilon & (3) \\ \forall i \in Dom(P) : \Gamma, F, S, i \vdash P : \{\!|\sigma, m, \alpha \rightarrow \gamma|\!\}_M & (4) \\ \forall i \in Dom(P) : \Gamma, F, S \vdash H[i].handles\ P & (5) \end{array}}{\Gamma, F, S \vdash \langle P, H \rangle :: \{\!|\sigma, m, \alpha \rightarrow \gamma|\!\}_M}$$

2.2. Un ejemplo de la ejecución del bytecode en la JVM

Este ejemplo ha sido sacado de [16].

```
javac Employee.java
javap -c Employee < Employee.bc
```

El bytecode de la clase Employee.java se puede observar en el cuadro 2.1. Esta clase es muy sencilla. Contiene una constructora y tres métodos.

Se puede apreciar que algunas instrucciones de bytecode tienen el prefijo 'a' o 'i'. Por ejemplo, en la constructora de la clase Employee se puede ver `aload_0` y `iload_2`. El prefijo es representativo del tipo de instrucción de bytecode que se está utilizando. El prefijo `'a'` significa que el opcode está manipulando un objeto. El prefijo `'i'` significa que la instrucción está manipulando un entero. Otras instrucciones usan `'b'` para byte, `'c'` para caracteres, `'d'` para double, etc. Este prefijo da un conocimiento inmediato sobre el tipo de

datos que se está manipulando.

Para entender los detalles del bytecode, es importante conocer como trabaja la máquina virtual de Java (JVM) respecto a la ejecución del bytecode. JVM es una máquina de pila. Cada hebra tiene una pila JVM que almacena marcos. Se crea un marco cada vez que se invoca un método. Un marco consiste en una pila de operandos, un array de variables locales y un puntero a la *constant pool* de la clase del método actual, que es un almaceb de todos los atributos de la clase.

Conceptualmente, un posible esquema sería el mostrado en la figura 2.1

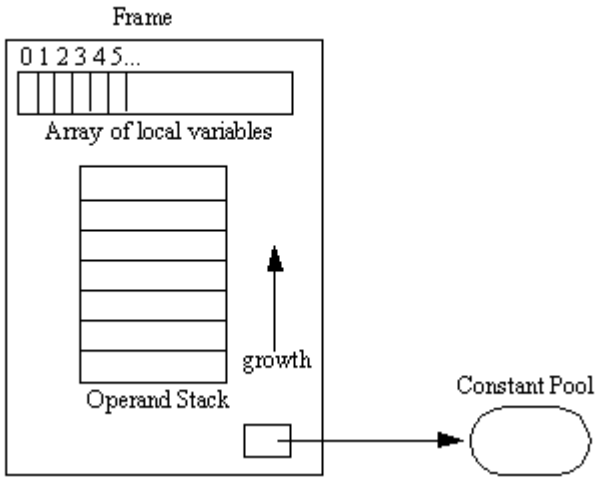


Figura 2.1: Marcos o registros de activación de la JVM

El array de variables locales, también llamado la tabla de variables locales, contiene los parámetros del método y también se usa para almacenar los valores de las variables locales. Los parámetros se almacenan primero, comenzando en el índice 0. Si el marco es para una constructora o para un método instanciado la referencia se almacena en la posición 0. Después, la posición 1 contiene el primer parámetro formal, la posición 2 el segundo, y así sucesivamente. Para un método estático, el primer parámetro formal se almacena en la posición 0, el segundo en la posición 1, etc...

El tamaño del array de variables locales se determina en tiempo de compilación y es dependiente del número y del tamaño de las variables locales y de los parámetros formales del método en cuestión. La pila de operandos es

una pila LIFO(Last In First Out) utilizada para instroducir y sacar valores. Su tamaño se determina también en tiempo de compilación. Algunas instrucciones introducen valores en la pila de operandos, otros cogen operandos de la pila, los manipulan, y vuelven a introducir el resultado. La pila de operandos también se usa para albergar valores devueltos por los métodos.

En el cuadro 2.2 se muestra el bytecode correspondiente al método employeeName().
El bytecode para este método consiste en tres instrucciones. La primera instrucción , `aload_0`, introduce el valor del índice 0 del array del variables locales dentro de la pila de operandos. La referencia `this` siempre se almacena en la posición 0 del array de variables locales para las constructoras y para los métodos instanciados.

La siguiente instrucción, `getfield`, se usa para buscar un campo de un objeto. Cuando se ejecuta esta instrucción, la cima de la pila , `this`, se saca. Después, #5 se usa para construir un índice dentro del *constant pool* de la clase donde la referencia a `name` se almacena. Cuando esta referencia se busca, esta se introduce dentro de la pila de operandos.

La última instrucción, `areturn`, devuelve una referencia de un método. Más especificamente, la ejecución de `areturn` causa que la cima de la pila de operandos, la referencia a `name`, sea sacada e introducida dentro de la pila de operandos del método invocado.

El método `employeeName` es muy simple. Sin embargo, se necesita examinar los valores que aparecen a la izquierda de cada opcode. En el bytecode del método `employeeName` estos valores son 0, 1, y 4. Cada método tiene su correspondiente array de bytecode. Estos valores corresponden al índice dentro del array donde cada instrucción y sus argumentos se almacenan. Uno se podría preguntar por qué estos valores no son secuenciales, de hecho, el bytecode se llama así porque cada instrucción ocupa un byte, entonces ¿por qué no son los índices 0, 1, y 2?. La razón es que algunos de las instrucciones tienen parámetros que ocupan mucho lugar dentro del array de bytecode. Por ejemplo, `aload_0` no tiene parámetros y naturalmente ocupa un byte en el array de bytecode. Por lo tanto, la siguiente instrucción, `getField`, está en la posición 1. Sin embargo, `areturn` se encuentra en la posición 4. Esto es debido a que la instrucción `getField` y sus parámetros ocupan las posiciones 1, 2 y 3. La posición 1 se utiliza para `getField`, la posición 2 y la 3 se utilizan para almacenar los parámetros. Con estos parámetros se construye un índice para la *constant pool* de la clase donde el valor se almacena.

El diagrama de la figura 2.2 muestra como es el aspecto del array de bytecode del método employeeName.

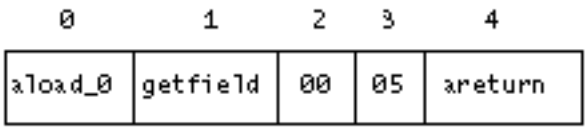


Figura 2.2: El array de bytecode 1 para metodo employeeName()

En realidad, el array de bytecode contiene bytes que representan las instrucciones. En un archivo .class, se observan los valores de la figura 2.3 dentro del array de bytecode.

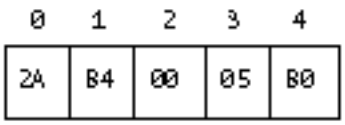


Figura 2.3: El array de bytecode 2 para metodo employeeName()

2A , B4 , and B0 corresponden a `aload_0`, `getfield`, y `areturn` respectivamente.

En el cuadro 2.3 se muestra el bytecode de la constructora `Employee()`.
La primera instrucción en la posición 0, `aload_0`, introduce la referencia `this` dentro de la pila de operandos. La siguiente instrucción en la posición 1, `invokespecial`, llama a la constructora de la superclase. Esto se debe a que aunque no se haga explícito hereda de `java.lang.Object`, y el compilador provee el byecode necesario para invocar la constructora de la superclase. Durante esta instrucción, la cima de la pila de operandos, `this`, se saca. Las dos siguientes instrucciones, en las posicines 4 y 5 introducen los valores de las dos primeras posiciones del array de variables locales dentro de la pila de operandos. El primer valor introducido es la referencia this. El segundo valor es el primer parámetro formal de la constructora, `strName`. Estos valores son introducidos para preparar la ejecución de la instrucción `putfield` que se almacena en la posición 6.

`Putfield` se saca la cima y la subcima de la pila de operandos y almacena una referencia a `strName` dentro del `name` del objeto referenciado por `this`. Las tres siguientes instrucciones situadas en 9, 10 y 11, ejecutan la misma operación con el segundo parámetro formal de la constructora, `num`, y la variable instanciada, `idNumber`. Los tres siguientes instrucciones situadas en las posiciones 14, 15, y 16 preparan la pila para la llamada al método `storeData`. Estas instrucciones meten la referencia `this`, `strName`, y `num`, respectivamente. La referencia `this` se debe introducir porque se está llamando a un método instanciado. Si el método hubiese sido declarado estático, no hubiese sido necesario introducir la referencia `this`. Los valores strName y num son introducidos porque son parámetros del método `storeData`. Cuando el método storeData se ejecute, `this`, `strName`, y `num`, ocuparán los índices 0,1 y 2, respectivamente, del array de variables locales contenido dentro del marco de este método.

```
Compiled from Employee.java
Class Employee extends java.lang.Object{
 public Employee (java.lang.String, int);
 public java.lang.String employeeName();
 public int employeeNumber();
}

Method Employee(java.lang.String, int)
0 aload_0
1 invokespecial #3
< Method java.lang.Object() >
4 aload_0
5 aload_1
6 putfield #5
< Field java.lang.String() name >
9 aload_0
10 iload_2
11 putfield #4 < Field int idNumber >
14 aload_0
15 aload_1
16 iload_2
17 invokespecial #6 < Method void storeData(java.lang.String, int) >
20 return

Method java.lang.String employeeName()
0 aload_0
1 getfield #5 < Field java.lang.String name >
4 areturn

Method java.lang.String employeeNumber()
0 aload_0
1 getfield #4 < Field int idNumber >
4 ireturn

Method void storeData(java.lang.String, int)
0 return
```

Cuadro 2.1: El bytecode de los métodos de la clase Employee.java

```
public String employeeName(){
 return name;
}

Method java.lang.String employeeName()
0 aload_0
1 getField #5 < Field java.lang.String name >
4 areturn
```

Cuadro 2.2: El bytecode del método employeeName()

```
public Employee (String strName, int num){
  name = strName;
  idNumber = num;
  storeData(strName, num);
}

Method Employee(java.lang.String, int)
0 aload_0
1 invokespecial #3 < Method java.lang.Object()>
4 aload_0
5 aload_1
6 putField #5 < Field java.lang.Object() >
9 aload_0
10 iload_2
11 putField #4 < Field int idNumber >
14 aload_0
15 aload_1
16 iload_2
17 invokespecial #6 < Method void storeData(java.lang.String,int) >
20 return
```

Cuadro 2.3: El bytecode de la constructora Employee()

Capítulo 3

Generación de ecuaciones de coste a partir del bytecode de Java

El objetivo de este capítulo es describir el desarrollo de un enfoque automático del análisis de coste del bytecode de Java que genera de forma estática relaciones de coste [19]. Estas relaciones definen el coste de un programa en función del tamaño de sus argumentos de entrada.

Un lenguaje orientado a objetos de bajo nivel como el bytecode de Java introduce nuevos desafíos, principalmente debido a: 1) su desestructurado grafo de control; 2) sus características de lenguaje orientado a objetos; 3) su modelo basado en una pila donde se almacenan los valores intermedios.

Este proceso tiene como entrada el bytecode correspondiente al método y produce unas relaciones de coste después de ejecutar los siguientes pasos:

1. Primeramente, el bytecode se transforma en un grafo de control de flujo (CFG). Esto permite hacer explícito el flujo de control desestructurado.

2. Después, el CFG se representa como un conjunto de reglas usando una representación recursiva intermedia en la cual se aplana la pila local convirtiendo su contenido en una seria de variables locales adiccionales.

3. En el tercer paso, se infieren las relaciones de tamaño entre las variables de entrada para todas las llamadas que están en las reglas por medio

de un análisis estático.

4. El cuarto paso aporta a cada regla de la representación recursiva una aproximación segura del conjunto de argumentos de entrada que son 'relevantes' para el coste. Esto se lleva a cabo un usando análisis estático simple, denominado *slicing* [22].

5. A partir de la representación recursiva, sus argumentos relevantes, y sus relaciones de tamaño, el quinto paso produce, automáticamente, como salida la relación de coste que expresa el coste del método en función de sus argumentos de entrada.

En la figura 3.1 se muestra un esquema de los pasos a seguir en el análisis del coste del bytecode de Java.

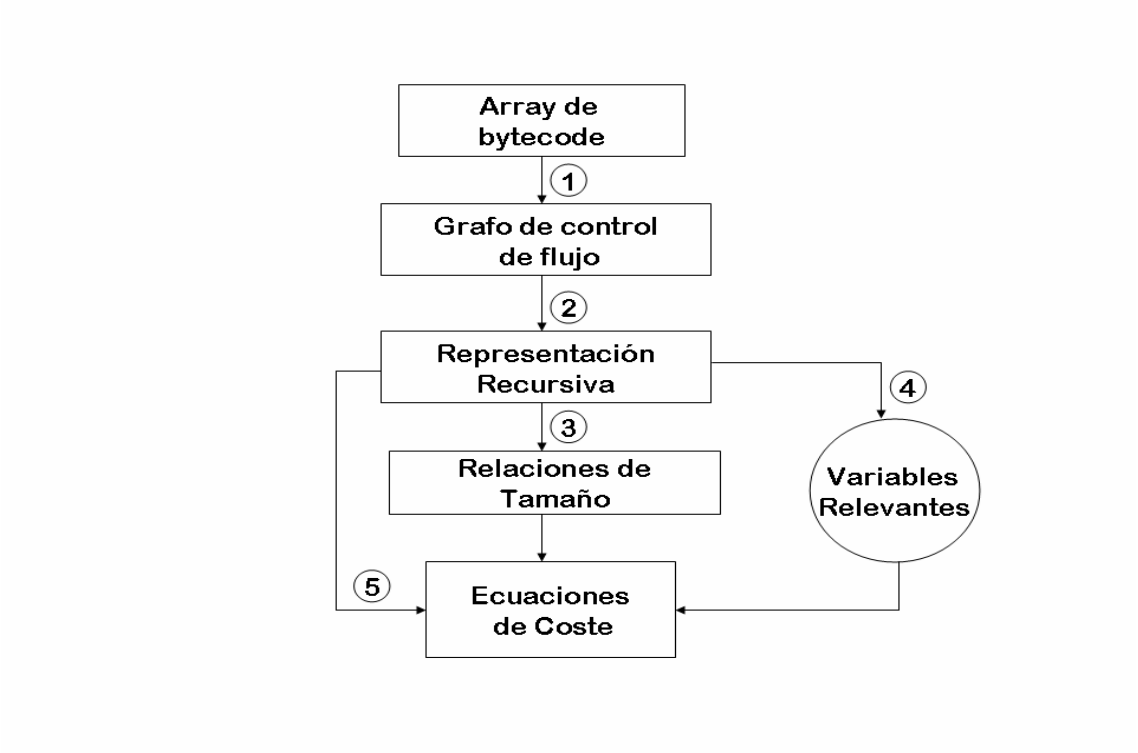


Figura 3.1: Visión de conjunto del análisis del coste de bytecode de Java

Brevemente y a partir de un ejemplo, se explican las diferentes fases del análisis del coste. El ejemplo que se utiliza se corresponde con la implementación recursiva de la sucesión de Fibonacci y se muestra en el cuadro 3.1.

```
class Fibonacci{
 static int fibonacciMethod(int n){
  if ((n == 0) || (n == 1)){
   return 1;
  }
  else{
   return (fibonacciMethod(n - 1) + fibonacciMethod(n - 2));
  }
 }
}
```

Cuadro 3.1: Método de Fibonacci

Dado un número natural n, la llamada `fin(n)` calcula el n-ésimo término en la sucesión de Fibonacci. El bytecode de entrada al análisis del coste se muestra en el cuadro 3.2.

La variable de entrada n se almacena en la variable local con índice 0. Esta variable local se compara con las constantes 0 y 1 (los casos base de la serie de Fibonacci). Si cualquiera de las dos comparaciones tienen éxito, la ejecución salta a la instrucción 9 donde la constante 1 se introduce en la pila y se devuelve como el resultado del método. En caso contrario, cuando las dos comparaciones fallan, el control se dirige hacia la instrucción 11 donde el método se llama recursivamente dos veces con valores n - 1 y n - 2, respectivamente. Los valores obtenidos se suman, y así de esta manera se obtiene el valor de retorno del método.

El análisis del coste empieza a partir de este bytecode y lleva acabo cinco importantes pasos que se describen en los apartados siguientes.

```
0: iload_0
1: ifeq 9
4: iload_0
5: iconst_1
6: if_icmpne 11
9: iconst_1
10: ireturn
11: iload_0
12: iconst_1
13: isub
14: invokestatic #2
17: iload_0
19: isub
20: invokestatic #2
23: iadd
24: ireturn
```

Cuadro 3.2: El bytecode del método de Fibonacci

3.1. Grafos de control de flujo

Esta sección describe la generación de un grafo de control de flujo (CFG, Control Flow Graph) a partir del bytecode del método. Esta técnica se basa en ideas de compiladores [2] y [1] que actualmente se aplican en el análisis del bytecode de Java.

Dado un método m, se denota como G_m su CFG que es un grafo dirigido donde cada nodo representa un bloque. Cada bloque $Block_{id}$ es una tupla de la forma <id, G, B, D> donde: id es el identificador del bloque; G es el guarda del bloque que indica las condiciones bajo las que el bloque se ejecuta; B es una secuencia de instrucciones de bytecode contiguas para las cuales se garantiza que serán ejecutadas sin ningún tipo de condición (por ejemplo, si G tiene éxito entonces todas las instrucciones en B serán ejecutadas antes de que el control se diriga a otro bloque); y D es la lista de adyacencia para el bloque $Block_{id}$, es decir si id'pertenece al conjunto D entonces hay un arco de $Block_{id}$ a $Block_{id'}$. Los guardas tienen la forma de guard(C), donde C es una condición booleana sobre las variables locales y los elementos de la pila.

Una gran parte de las instrucciones de bytecode tienen un único sucesor. Sin embargo, existen tres tipos de bifurcaciones:

Saltos condicionales: de la forma 'pc_i: if X pc_j'. Dependiendo de si es cierta o no la condición, la ejecución puede saltar a pc_j o continuar por pc_{i+1}. El grafo describe este comportamiento a partir de dos arcos desde el bloque que contiene la instrucción pc_i, al que empieza por la instrucción pc_j y al que empieza por pc_{i+1}. Cada uno de estos nuevos bloques empiezan con un guarda que expresa la condición bajo la cual ese bloque se ejecuta.

Enlace dinámico: de la forma 'pc_i: invokevirtual c.m'. El tipo del objeto 'o' cuyo método se está invocando no se conoce estáticamente (podría ser c o cualquier subclase de c). [11]Por ejemplo, considérese el ya familiar ejemplo de una jerarquía de clases que hace uso del polimorfismo. El tipo genérico es de la clase base Figura, y los tipos específicos derivados son Circulo, Cuadrado y Triángulo. La meta normal en la programación orientada a objetos es que la mayor cantidad posible de código manipule referencias de la clase base (en este caso, Figura) de forma que si se decide extender el programa añadiendo una clase nueva (Romboide, derivada de Figura, por ejemplo), la gran mayoría de código no se vea afectada. En este ejemplo, el método asignado estáticamente en la interfaz figura es dibujar(), por tanto, se pretende que el programador cliente invoque dibujar() a través de una referencia a Figura genérica. El método dibujar() está superpuesto en todas las clases derivadas, y dado que por ello es un método de correspondencia dinámica, se obtendrá el comportamiento adecuado incluso aunque se invoque a través de una referencia a Figura genérica. Esto es el polimorfismo.

Por consiguiente, se suele crear un objeto específico (Circulo, Cuadrado o Triángulo), se aplica un molde hacia arriba a una Figura (olvidando el tipo específico del objeto), y se usa como una referencia Figura anónima en el resto del programa.

Para dar un breve respaso al polimorfismo y aplicar un molde hacia arriba, véase el ejemplo del cuadro 3.3.
La clase base contiene un método dibujar() que usa indirectamente toString() para imprimir un identificador de la clase pasando this a System.out.println(). Si esta función ve un objeto, llama automáticamente al método toString() para producir una representación String. Cada una de las clases derivadas superpone el método toString() (de Object) de forma que dibujar() acaba imprimiendo algo distinto en cada caso. En el método main() se crean tipos específicos de Figura que después se añaden a un ArrayList. Este es el mo-

```
class Figura {
 void dibujar(){
  System.out.println(this + '.dibujar()');
 }
}

class Circulo extends Figura {
 public String tostring() { return 'Círculo';}
}

class Cuadrado extends Figura {
 public String tostring() { return 'Cuadrado';}
}

class Triangulo extends Figura {
 public String tostring() { return 'Triangulo';}
}

public class Figura{
 public static void main(String[] args){
  ArrayList s = new ArrayList();
  s.add(new Circulo());
  s.add(new Cuadrado());
  s.add(new Triangulo());
  Iterator e = s.iterator();
  while (e.hasNext())
   ((Figura)e.next()).dibujar();
  }
}
```

Cuadro 3.3: La clase Figura.java

mento en el que se aplica un molde hacia arriba puesto que ArrayList sólo guarda Objects. En el momento de recuperar un elemento de ArrayList con next(), todo se vuelve más complicado. Dado que ArrayList simplemente guarda Objects, naturalmente next() produce una referencia Object. Pero se sabe que verdaderamente es una referencia a Figura. Por lo tanto es necesaria una con versión '(Figura)'. En este caso la conversión es solo parcial pues no

se llega hasta Circulo, Cuadrado y Triangulo. Esto se debe a que lo único que se sabe en este momento es que ArrayList está lleno de Figuras. En tiempo de compilación, se refuerza esto sólo por reglas autoimpuestas; en tiempo de ejecución prácticamente se asegura.

Ahora toma su papel el polimorfismo y se determina el método exacto invocado para Figura para saber si es una referencia a Circulo, Cuadrado o Triangulo.

Por lo tanto, no se puede determinar estáticamente que método se va a invocar. Así pues, se necesita hacer explícito en el grafo todas las posibilidades. Para tratar el problema del enlace dinámico, se utiliza la función resolve_virtual(c, m) que devuelve el conjunto de MétodosResueltos en parejas de <d, $\{c_1, ..., c_k\}$>, donde d es una clase que define el método con nombre m y cada c_i es c o una subclase de c que hereda ese específico método de d. Para cada <d, $\{c_1, ..., c_k\}$> perteneciente al conjunto de MetodosResueltos se genera un nuevo bloque $bloque_d^{pci}$ con una única instrucción invoke(d:m) la cual da soporte para una invocación no virtual de m que está definido en la clase d. Además, el bloque tiene un guarda de la forma instanceof(o,$\{c_1$, ..., $c_k\}$) (o es un elemento de la pila) para indicar que el bloque solo se ejecuta en el caso de que o sea una instancia de una de las clases c_1, ..., c_k. Un arco desde el bloque que contiene pc_i hasta el bloque $bloque_d^{pci}$ se añade, junto con un arco desde $bloque_d^{pci}$ hasta el bloque que contiene la siguiente instrucción pc_{i+1} (que describe el resto de la ejecución después de llamar a m).

Excepciones: Por lo que se refiere a la estructura del CFG, las excepciones no se tratan de una manera especial. En cambio, la posibilidad de que ocurra una excepción durante la ejecución de una instrucción de bytecode b, se trata simplemente como una bifurcación después de b. Se permite que el $Block_b$ termine con b; los arcos que salen de $Block_b$ se generan de forma normal, y llegan a alcanzar los subgrafos que corresponden a los manejadores de excepciones.

3.1.1. Ejemplo 1

La ejecución del método `add(n,o)` que se muestra en el cuadro 3.4 calcula: $\Sigma_{i=0}^{n}$ si o es una instancia de A; $\Sigma_{i=0}^{[n/2]} 2i$ si o es una instancia de B; y $\Sigma_{i=0}^{[n/3]} 3i$ si o es una instancia de C. En los cuadros: 3.5, 3.6, 3.7 3.8 se muestra el bytecode asociado al código Java. El CFG del método `add` se muestra se muestra en la figura 3.2. El hecho de que el sucesor de 6: `if_icmpgt 16`

pueda ser la instrucción 7 o la 16 se expresa a través de dos arcos desde el bloque $BLock_1$, uno al bloque $Block_2$ y otro al bloque $Block_3$, a través de los guardas `icmpgt` al bloque $Block_2$ y `icmple` al bloque $Block_3$. La invocación `13: invokevirtual A.incr: (I)I` se divide en tres posibles escenarios de ejecución descritos en los bloques $Block_4$, $Block_5$, y $Block_6$. Dependiendo del tipo del objeto o (la subcima de la pila, denotado por `s(top(1))`) en los guardas) solo se ejecutará uno de estos bloques dependiendo y por lo tanto solo se invocará una de las definciones de `incr`. Nótese que `invokevirtual` se ha sustituido por `resolve_virtual`. El comportamiento de la excepción cuando o es *NULL* se describe en los bloques $Block_7$ y $Block_{exc}$.

```
class A{
  int incr (int i){
    return i+1;}};
class B extends A{
  int incr (int i){
    return i+2;}};
class C extends AB{
  int incr (int i){
    return i+3;}};
Class Main{
  int add (int n, A o){
    res=0;
    i=0;
    while(i ≤ n){
      res = res + 1;
      i=o.incr(i);}
    return res;}};
```

Cuadro 3.4: código Java

```
0: iload_1
1: iconst_1
2: iadd
3: ireturn
```

Cuadro 3.5: El bytecode asociado al método `incr` de la clase A

```
0: iload_1
1: iconst_2
2: iadd
3: ireturn
```

Cuadro 3.6: El bytecode asociado al método `incr` de la clase B

```
0: iload_1
1: iconst_3
2: iadd
3: ireturn
```

Cuadro 3.7: El bytecode asociado al método `incr` de la clase c

```
0: iconst_0
1: istore_3
2: iconst_0
3: istore_4
4: iload_4
5: iload_1
6: if_icmpgt 16
7: iload_3
8: iload_4
9: iadd
10: istore_3
11: aload_2
12: iload_4
13: invokevirtual A.incr:(I)I
14: istore_4
15: goto 4
16: iload_3
17: ireturn
```

Cuadro 3.8: El bytecode asociado al método `add`

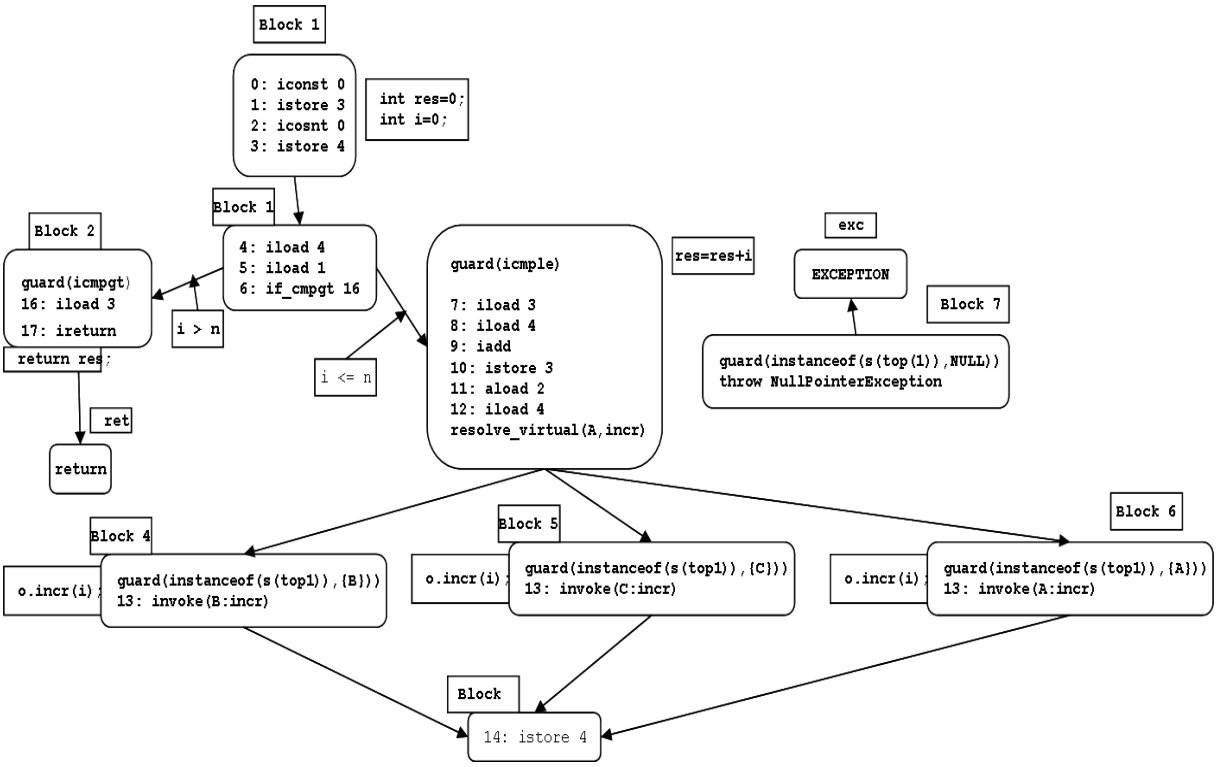


Figura 3.2: Grafo de control de flujo para el método `add`

3.1.2. Ejemplo 2

Considerando el CFG de la figura 3.3.
Cada bloque contiene un identificador (Block i), opcionalmente un guarda, y una secuencia de instrucciones de bytecode para las cuales se garantiza que van a ser ejecutadas secuencialmente. El bloque inicial es el bloque 0. Bifurcaciones originadas por excepciones, saltos condicionales o el enlace dinámico se controlan a partir de los guardas de la forma guard(C), los cuales indican las condiciones necesarias para que se ejecute un determinado bloque. Por ejemplo, el bloque 1 y el bloque 2 contienen guard(ifqe) y guard(ifne), respectivamente. Por lo tanto, si cuando se está ejecutando el bytecode 1 la cima de la pila (ifqe 9) es igual a 0, la ejecución se dirige al bloque 1, sin embargo se dirige al bloque 2 si la cima no es igual a 0. Es importante señalar que los guardas no se tienen en cuenta para calcular el coste de un programa, ya que estos no son parte original del programa. Los guardas aportan información muy relevante en la generación de unas ecuaciones de coste veraces, precisas y rigurosas.

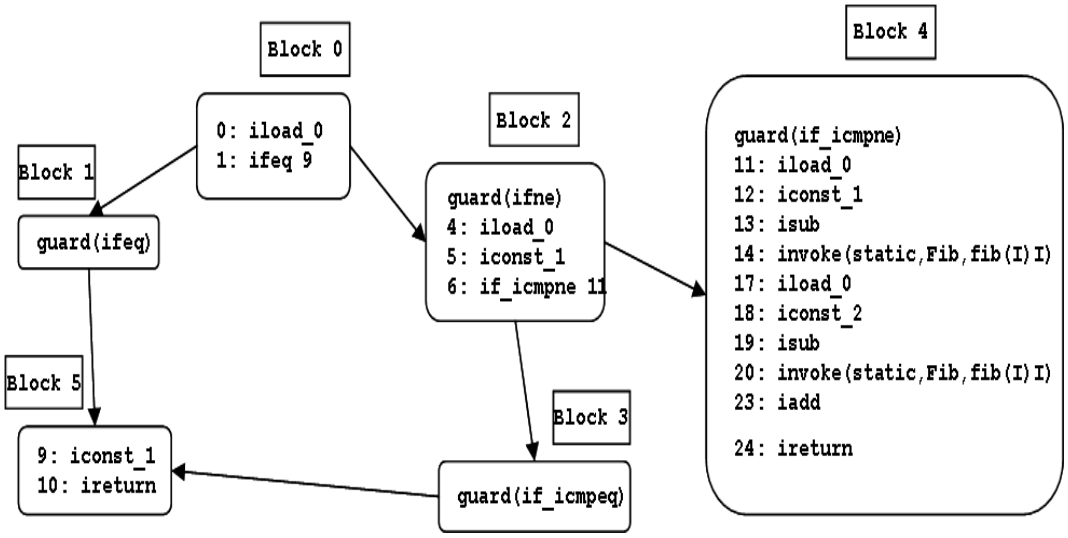


Figura 3.3: Grafo de control de flujo para el método Fibonacci

3.2. Representación recursiva

En esta sección se presenta un método para obtener una representación recursiva del bytecode de un método, donde 1) Las iteraciones se convierten en recursión y 2) la pila de operandos se aplana en el sentido de que su contenido se representa como una serie de variables locales. Tener en cuenta que esto es posible porque, en cada punto del programa, la altura de la pila puede ser determinada estáticamente.

Suponemos que m es un método definido en la clase c, con variables locales $\overline{l_k} = l_0, ..., l_k; l_0$ contiene un puntero al objeto this, l_1, ..., l_n son los n argumentos de entrada del método, y l_{n+1}, ..., l_k corresponden a las k - n variables locales declaradas en m. Además de estos argumentos, se añaden las variables $\overline{s_t} = s_0$, ..., s_{t-1}, las cuales corresponden a los elementos que están en la pila, siendo s_0 la posición más baja. Como notación h_{id} representa la altura de la pila al entrar al bloque $Block_{id}$, y $s_t^-|h_{id}$ denota la restricción de $\overline{s_t}$ para la correspondiente pila (se refiere a las variables de la pila que se utilizan). La representación recursiva de m se define como un conjunto de reglas `head` $\leftarrow$ `body` obtenido a partir de su grafo de control de flujo G_m de esta manera:

1. La regla a la entrada del método es c:m($\overline{l_n}$, ret) $\leftarrow$ c:m^0($\overline{l_k}$, ret), donde ret es una variable para almacenar el valor de retorno.

2. Para cada $Block_{id}$ = <id, G, $\overline{B_p}$, {id_1, ..., id_j}> perteneciente a G_m se genera una regla:

$$c : m^{id}(\overline{l_k}, \overline{s_t}|h_{id}, ret) \leftarrow G', \overline{B_p}'(call_{id_1}; ...; call_{id_j})$$

   donde:

   - G es el guarda del bloque y que $\overline{B_p}$ es la secuencia de bytecodes del bloque que se ejecutan si se satisface el guarda del bloque, G' $\cup$ $\overline{B_p}$' se obtiene a partir de G $\cup$ $\overline{B_p}$, y $call_{id_1}$; ...; $call_{id_j}$ son las posibles llamadas a los bloques.

   - Cada $b_i \in G \cup \overline{B_p}$ se transforma en b_i' añadiendo explícitamente las variables (variables locales o variables de la pila) utilizadas como argumentos por b_i. Por ejemplo, iadd se transforma en iadd(s_{j-1}, s_j, s'_{j-1}), donde j es el índice de la cima de la pila justo antes de ejecutar iadd. Nótese que nos referimos dos veces al elemento de la pila j-1 llamándolo de dos maneras distintas: s_{j-1} se refiere al valor de entrada y s'_{j-1} se refiere al valor de salida.
     En el cuadro 3.9 se presenta la función de transformación para los bytecodes seleccionados.

La función `translate` para `iadd` funciona de la siguiente manera: La función coge como entrada el nombre del método actual, el contador de programa pc del bytecode, el bytecode (en este caso iadd), los nombres de las variables locales actuales $\overline{l_k}$ y los nombres de las variables de la pila actuales $\overline{s_t}'$. En la línea 1 se recupera el índice de la cima de la pila antes de ejecutar el bytecode actual. En la línea 2 se generan nuevos nombres para las variables de la pila $\overline{s_t}'$ renombrando las variables de salida de iadd. Como notación, dada una secuencia de a_n^- de elementos, $\overline{a_n}[i \longmapsto b]$ denota la sustitución en $\overline{a_n}$ del elemento a_i por b. En la línea 3 de devueve (ret <>) el bytecode transformado junto con los nuevos nombres de las variables de la pila. Si se asume que G = $pc_0 : b_0$ y $\overline{B_p} = \langle pc_1 : b_1, ..., pc_p : b_p \rangle$. La transformación del bytecode se hace iterativamente así:

`for i = 0 to p` $\langle b'_i, l_k^{-i+1}, s_t^{-i+1} \rangle$ = `translate` $(m, pc_i, b_i, \overline{l_k^i}, \overline{s_t^i})$

`translate(m, pc, iadd,` $\overline{l_k}, \overline{s_t}$`) :=`
`let j =  top_stack_index(pc, m) in`
$\overline{s_t}' =$ `` `` $\overline{s_t}[j-1 \longmapsto s'_{j-1}]$
`ret`$\langle iadd(s_{j-1}, s_j, s'_{j-1}), \overline{l_k}, \overline{s_t}' \rangle$

`translate(m, pc, guard(icmpgt),` $\overline{l_k}$`,` $\overline{s_t}$`)`
`let j = top_stack_index(pc, m) in`
`ret` $\langle guard(icmpgt(s_{j-1}, s_j)), \overline{l_k}, \overline{s_t} \rangle$

`translate(m, pc, ireturn(v),` $\overline{l_k}$`,` $\overline{s_t}$`)` `ret`$\langle ireturn(s_0, ret), \overline{l_k}, \overline{s_t}' \rangle$

`translate(m, pc, iload(v),` $\overline{l_k}$`,` $\overline{s_t}$`)`
`let j = top_stack_index(pc, m) in`
$\overline{s_t}' = \overline{s_t}[j+1 \longmapsto s'_{j+1}]$
`ret`$\langle iload(l_v, s'_{j+1}), \overline{l_k}, \overline{s_t} \rangle$

`translate(m, pc, invoke(b:m'),` $\overline{l_k}$`,` $\overline{s_t}$`) :=`
`let j = top_stack_index(pc, m),`
`n = number_of_arguments(b, m') in`
$\overline{s_t}' = \overline{s_t}[j-n \longmapsto s'_{j-n}]$
`ret`$\langle b : m'(s_{j-n}, ..., s_j, s'_{j-n}), \overline{l_k}, \overline{s_t}' \rangle$

Cuadro 3.9: Tranformación de las instrucciones de bytecode

Se comienza a partir de un conjunto inicial de variables locales y de pila, $\overline{l_k^0} = \overline{l_k}$ y $\overline{s_t^0} = \overline{s_t}$; en cada paso, `translate` coge como entrada los nombres de las variables de entrada y de pila que fueron generadas transformando el bytecodo previo. Al final de este bucle, se puede definir cada $call_{id_i}$, $1 \leq i \leq j$, como $c : m^{id_i}(\overline{l_k^{p+1}}, \overline{s_t^{p+1}} | h_{id}, ret)$.

3.2.1. Ejemplo 1

Consideremos el CFG de la figura 3.2. La traducción del Bloque $Block_3$ al bloque $Block_4$ funciona de como se describe en el cuadro 3.2.1

$add^3(\overline{l_4}, s_0, s_1, ret) \leftarrow$
`guard(icmple(`s_0`,` s_1`)),`
`iload(`l_3`,` s_0`'), iload(`l_4`,` s_1`'), iadd(`s_0`',` s_1`',` s_0`'')`
`istore(`s_0`'',` l_3`'), aload(`l_2`,` s_0`'''), iload(`l_4`,` s_1`''),`
`resolve_virtual(A, incr)`
`(`add^4`(`l_0`,` l_1`,` l_2`,` l_3`',` l_4`,` s_0`''',` s_1`'',` `ret);`
add^5`(`l_0`,` l_1`,` l_2`,` l_3`',` l_4`,` s_0`''',` s_1`'',` `ret);`
add^6`(`l_0`,` l_1`,` l_2`,` l_3`',` l_4`,` s_0`''',` s_1`'',` `ret))`

$add^4(\overline{l_4},$ `` s_0, s_1, ret) \leftarrow ``
`guard(instanceof(`s_0`, {B})),`
`B: incr(`s_0`,` s_1`,` s_0`'),`
$add^8(\overline{l_4}, s_0', ret)$.

En la regla add^3, el enlace dinámico se representa como una disyunción de llamadas a add^4. add^5, o add^6. Por lo tanto, en la regla add^4 encontramos una llamada a `incr` de la clase B que correponde a la traducción de `invoke(B:incr)`; los argumentos que se le pasan a `incr` son la cima y la subcima de la pila, el valor devuelto es el último argumento, se le pone un guión (s_0') porque es el valor actualizado.

3.2.2. Ejemplo 2

Como se ha especificado anteriormente, en esta representación cada bloque en el grafo se representa como una regla. La representación recursiva del método de fibonacci se muestra en el cuadro 3.10.

Como el método de Fibonacci es recursivo, la importancia de esta representación no se advierte totalmente. La notación $fib_i(\bar{x}) \leftarrow B_i$ significa que la ejecución del bloque $Block_i$ sobre las variables de entrada $\bar{x}$ consiste en las acciones contenidas en B_i. Los guardas y las llamadas a los bloques se representan explícitamente, pero por simplicidad el resto del bytecode se escribe como `BC(`$Block_i$`)`. El operador ';' significa disyunción. La regla principal es fib_a. El exponente sólo se usa para distinguir diferentes llamadas a la regla principal dentro del cuerpo de la misma regla, como en el caso del la regla

fib^a(n) $\leftarrow$ BC($BLock_0$),
(fib_1(n', s_0); fib_2(n', s_0)).

$fib_1(n, s_0) \leftarrow$ guard(ifeq(s_0)),
BC($Block_1$), fib_5(n').

fib_2(n, s_0) $\leftarrow$ guard(ifne(s_0)),
BC($Block_2$), (fib_3(n', s_0', s_1'); fib_4(n', s_0', s_1')).

fib_3(n, s_0, s_1) $\leftarrow$ guard(if_icmpeq(s_0, s_1)),
BC($Block_3$), fib_5(n').

fib_4(n, s_0, s_1) $\leftarrow$ guard(if_icmpne(s_0, s_1)),
BC($Block_4$), fib^b(n'), fib^c(n'').

fib_5(n) $\leftarrow$ BC($Block_5$).

Cuadro 3.10: Representación recursiva intermedia del método de Fibonacci

fib_4 con fib^b y fib^c.

En el ejemplo, aparte del parámetro n, se tienen que tener en cuenta dos variables mas: los elementos s_0 y s_1 de la pila (cuya máxima altura es 2). La regla $fib_2(n, s_0)$ modela el comportamiento del bloque 2 que se ejecutará si la cima de la pila es distinta de 0 (como se indica en guard(ifne(s_0))) y después de ejecutar BC($Block_2$), se puede seguir por $Block_3$ o por $Block_4$ (esto se hace con $fib_3(n', s_0', s_1'); fib_4(n', s_0', s_1'))$), dependiendo del resultado de los guardas al comienzo de los dos bloques.

3.3. Relaciones de tamaño

Es indispensable obtener relaciones de tamaño entre los estados en diferentes puntos del programa para crear las relaciones de coste. En particular, son esenciales para definir el coste de un bloque en términos del coste de sus sucesores. En general, se pueden utilizar diversas medidas para determinar el tamaño de la entrada. En el bytecode de Java se consideran dos casos: para

variables enteras, las relaciones de tamaño son restricciones sobre los posibles valores que puede tomar la variable, para variables que son referencias, son restricciones a la longitud de la ruta de acceso más larga.

Por ejemplo, consideremos los dos bucles siguientes escritos en Java por mayor simplicidad:

```
(1) while (i > 0) i--;
(2) while (l != null) l = l.next;
```

Para el caso (1) una relación de tamaño útil para el análisis del coste es que i sea siempre mayor que 0 y que decrezca en 1 en cada iteración. Para el caso (2) que la ruta de acceso más larga desde l decrezca en 1 en cada iteración.

Inferir relaciones de tamaño no es un camino directo: tales relaciones bien podrían ser el resultado de ejecutar algunas declaraciones, llamadas a métodos o bucles.
Para crear el modelo de coste, se necesita, para cada regla de la representación recursiva las relaciones de tamaño entre las variables de la cabeza de la regla y las variables utilizadas en las llamadas (a reglas) que se encuentran en el cuerpo. Nótese que dada una regla $\mathrm{p}(\bar{\mathrm{x}}) \leftarrow \mathrm{G}$, B_k^-, $(q_1; ...; q_n)$, cada $b_i \in B_k^-$ puede ser una instrucción de bytecode o una llamada a otra regla. Se denota por calls($\overline{B_k}$) el cojunto d b_i que corresponden a una llamada un método y bytecode($\overline{B_k}$) al conjunto de b_i correspondientes a otros bytecodes.

Definición (llamadas a las relaciones de coste). Sea R_m la representación recursiva de un método m, donde cada regla adopta la forma $\mathrm{p}(\bar{\mathrm{x}})$ $\longleftarrow$ G, B_k^-, $(q_1(\bar{\mathrm{y}}); ...; q_n(\bar{\mathrm{y}}))$. Las llamadas a las relaciones de tamaño de R_m son de la forma:

$\langle \mathrm{p}(\bar{\mathrm{x}}), \mathrm{p'}(\bar{\mathrm{z}}), \psi \rangle$ donde $\mathrm{p'}(\bar{\mathrm{z}}) \in \mathrm{calls}(\overline{B_k}) \cup \{\mathrm{pcont}(\bar{\mathrm{y}})\}$

decribiendo, para todas las reglas, la relación de tamaño entre $\bar{\mathrm{x}}$ y $\bar{\mathrm{z}}$ cuando se llama $\mathrm{p'}(\bar{\mathrm{z}})$, donde $\mathrm{pcont}(\bar{\mathrm{y}})$ se refiere al punto del programa inmediatamente después de $\overline{B_k}$. La relación de tamaño ψ es da como una conjunción de restricciones lineales $a_0 + a_1 v_1 + ... + a_n v_n$ op 0, donde op $\in \{=, \leq, <\}$, cada a_i es una constante y $v_k \in \bar{\mathrm{x}} \cup \bar{\mathrm{z}}$ para cada k. Es importante tener en cuenta que no es necesario tener relaciones independientes para cada $q_i(\bar{\mathrm{y}})$ porque tienen exactamente las mismas variables.

De una manera sencilla, precisa y eficiente, el análisis de las relaciones de

tamaño para la representación recursiva de los métodos se puede hacer en dos pasos:

1. Compilando los bytecodes en las restricciones lineales que se imponen sobre las variables.

2. Computando un 'bottom-up fixpoint' sobre las reglas compiladas utilizando algoritmos estándars de 'bottom-up fixpoint'.

Se puede hacer la compilación a restricciones lineales a través de la función α_{size}, que lo que hace básicamente es reemplazar guardas y bytecodes por las restricciones que imponen a las correspondientes variables. En general, cada bytecode que ejecuta (de forma lineal) una operación aritmética se reemplaza por su correspondiente restricción lineal, y cada bytecode que manipula objetos se compila a unas restricciones lineales sobre la longitud de la ruta de acceso más larga a partir de la correspondiente variable. A continuación se muestran algunos ejemplos de la abstracción de guardas y de bytecodes en restricciones lineales:

(1) $\alpha_{size}(\texttt{iload}(l_1, s_0))$:= $(l_1 = s_0)$
(2) $\alpha_{size}(\texttt{iadd}(s_1, s_0, s'_0))$:= $(s'_0 = s_0 + s_1)$
(3) $\alpha_{size}(\texttt{guard}(\texttt{icmpgt}(s_1, s_0)))$:= $(s_1 > s_0)$
(4) $\alpha_{size}(\texttt{getfield}(s_1, f, s'_1))$:= $(s'_1 < s_1)$

3.3.1. Ejemplo 1

A partir de las reglas que corresponden a la representación recursiva del cuadro 3.2.1 resultan las llamadas a las relaciones de tamaño del cuadro 3.11.

$\langle add^3(l_0, l_1, l_2, l_3, l_4, ret), add^3\_cont(l_0, l_1, l_2, l'_3, l_4, ret), \{l_4 \leq l_1, l'_3 = l_3 + l_4\}\rangle$
$\langle add^4(l_0, l_1, l_2, l_3, l_4, ret), B : incr(l_2, l_4, ret), \{\}\rangle$
$\langle add^4(l_0, l_1, l_2, l_3, l_4, ret), add^4\_cont(l_0, l_1, l_2, l_3, l_4, s'_0, ret), s'_0 = l_4 + 2\}\rangle$

Cuadro 3.11: Relaciones de tamaño del ejemplo de la figura 3.2.1

3.3.2. Ejemplo 2

Como se ha dicho antes, este paso consiste en inferir relaciones de tamaño entre los estados en diferentes puntos del programa. Concretamente, se infieren relaciones de tamaño entre las variables de entrada que se encuentran en la cabeza y las que se encuentran en el bloque o en llamadas a métodos dentro de su cuerpo. También se ha explicado con anterioridad que esto se hace a través de un cómputo 'bottom-up fixpoint' y que en en general se pueden utilizar diversas medidas para determinar el tamaño de un término de entrada y que afectarán a la precisión del resultado. Entre las medidas más utilizadas este sistema puede soportar: (1) valores enteros para variables numéricas (esto es, el tamaño de x es su valor) y (2) longitud de la ruta para punteros (por ejemplo, el tamaño de x es la longitud de la cadena más larga de punteros empezando desde x). Es importante remarcar que el tamaño de una variable es una pieza fundamental en la estimación del coste de los programas.

Se muestran las relaciones de tamaño para el método de fibonacci en el cuadro 3.14.

$\langle fib^a$(n) $\mapsto fib_1$(n', s_0), {n' = n, s_0 = n}$\rangle$

$\langle fib_0$(n) $\mapsto fib_2$(n', s_0), {n' = n, s_0 = n}$\rangle$

$\langle fib_1$(n, s_0) $\mapsto fib_5$(n'), {s_0 = 0, n = n'}$\rangle$

$\langle fib_2$(n, s_0) $\mapsto fib_3$(n', s_0', s_1'),
{$s_0 \neq$ 0, s_0' = n, s_1' = 1, n' = n}$\rangle$

$\langle fib_2$(n, s_0) $\mapsto fib_4$(n_0', s_0', s_1'),
{$s_0 \neq$ 0, s_0' = n, s_1' = 1, n' = n}$\rangle$

$\langle fib_3$(n, s_0, s_1) $\mapsto fib_5$(n'), {s_0 = s_1, n' = n}$\rangle$

$\langle fib_4$(n, s_0, s_1) $\mapsto fib^b$(n'), {$s_0 \neq s_1$, n' = n - 1}$\rangle$

$\langle fib_4$(n, s_0, s_1) $\mapsto fin^c$(n'), {$s_0 \neq s_1$, n' = n - 2}$\rangle$

Cuadro 3.12: Relaciones de tamaño del método de Fibonacci

En este ejemplo, el valor entero se utiliza como medida de tamaño. Por ejemplo, se puede observar que la tercera relación de tamaño se deriva de la segunda regla de la representación recursiva: las relaciones $\{s_0 = 0, n = n'\}$ significa que s_0 es 0 cuando se entra al bloque $Block_1$, como requiere el guarda, y n no se modifica dentro del bloque.

3.4. Variables relevantes

Consideremos un $Block_{id}$ dentro de un CFG representado por la regla c : $m^{id}(\overline{l_k}, ret) \leftarrow G, \overline{B_h}$, $(call_{id_1}; ...; call_{id_j})$. La función de coste para el bloque $Block_{id}$ tiene la forma C_{id}: $(Z)^n \rightarrow N_\infty$, con $n \leq k$, y donde Z es el conjunto de los todos los enteros y N_∞ es el conjunto de los número naturales aumentado con el símbolo ∞ que denota infinito.

El objetivo aqui es minimizar el número n de argumentos que se necesita tener en cuenta en las funciones de coste. Como es usual en el análisis de coste, se considera que el argumento de salida 'ret' no influye en el coste de ningún bloque, por lo tanto puede ser ignorado en las funciones de coste. Además, a veces es posible no tener en cuenta algunos argumentos de entrada.

Dada una regla, los argumentos que van a tener un impacto considerable en el coste de un programa son aquellos que podrían afectar directa o indirectamente a los guardas de los programas (esto es, pueden afectar al control del flujo del programa), o son utilizados como argumentos de entrada a métodos externos cuyo coste, a veces, puede depender del tamaño de su entrada. Para hacer esto, se necesita seguir las dependencias de los datos en contra del flujo de control y esto conlleva calcular un 'fixpoint'. Dada una regla $p(\bar{x}) \leftarrow$ body, $\hat{l_p} \subseteq \bar{x}$ es la sub-secuencia de variables relevantes para p. La secuencia $\hat{l_p}$, obtenida a partir de la unión de las secuencias $\{\hat{l_p}\}$ con $p \in P$ para un conjunto P de reglas, mantiene el orden de las variables.

3.4.1. Ejemplo 1

Como se ha dicho anteriormente, la información obtenida a partir de un análisis previo será usada para estimar el coste de los programas. El problema de resolver las ecuaciones de coste es, en general, muy difícil, y las herramientas disponibles que se utilizaron para probar el alcance del sistema denotaron varias limitaciones en lo que concierne a la clase de problemas que se tienen

que tratar. Por lo tanto, el propósito es intentar simplificar las cosas aplicando transformaciones a los resultados que se obtienen.

Como primer paso, uno se da cuenta que, en muchos casos, la pila sólo se utiliza para cargar un dato contenido en una variable que no es de pila y ejecutar por ejemplo una comparación; luego la posición de pila se vacia sin ningún tipo de modificación del dato cargado. En este caso, la variable de pila sólo se usa como un almacén temporal, y su uso termina justo después de la comparación; detectar este tipo de situaciones es posible en muchos casos y se busca unificar las variables que no son de la pila con la que si lo son en el sentido de eliminar la última para las relaciones. En la línea 1 del bytecode del método de fibonacci, aparece una comparación con 0 justo después de cargar n en la pila. Está claro, por lo tanto, que en la representación recursiva la variables s_0 puede ser reemplazada por n dentro del guarda, por consiguiente se tendrá `guard(ifeq(n))`. En el método Fib esta optimización permite eliminar las s_0 y s_1 en todas las relaciones; y n viene a ser la única variable que se necesita tener en cuenta.

Como ejemplo se puede decir que el índice de un bucle `for` es generalmente relevante porque este afecta al número de iteraciones; por otra parte aquella variable que se utiliza para almacenar resultados parciales no tiene efectos en el coste, a menos que su valor participe en cálculos cuyo tiempo de ejecución no sea fijo. Las variables relevantes resultan ser aquellas que están implicadas en los guardas o en llamadas a métodos, y esto se debe a:

1. Un guarda afecta al flujo de control de un programa, y por lo tanto a su tiempo de ejecución.

2. El coste de ejecutar métodos externos es claramente relevante en coste total.

3.5. Modelo de coste

A partir de este momento se define la función de coste C_{id}: $(Z)^n \rightarrow N_\infty$ para el bloque $Block_{id}$como un conjunto de ecuaciones de coste. La idea intuitiva es que dada una regla p($\bar{\text{x}}$) $\leftarrow$ G, B, (q_1, ..., q_n) asociada al $Block_{id}$ se genera:

1. Una ecuación de coste que define el coste de p a partir del coste de las declaraciones en B, mas el coste de su continuación, denotada por `pcont`.

2. Otra ecuación de coste que define el coste de pcont como el coste de o bien q_1 (si se satisface su guarda), ..., o bien el coste de q_n (si se satisface su guarda).

El coste de la continuación se especifica en una ecuación separada porque las condiciones para determinar el camino alternativo q_i que la ejecución tomará (con i = 1, ..., n) sólo se conoce al final de la ejecución de B; por lo tanto no puede ser evaluado antes de que B sea ejecutado. En la siguiente definición se utiliza la función α_{guard} para reemplazar aquellos guardas que indican el tipo de un objeto por el test adecuado (por ejemplo, α_{guard}(guard(instanceof(s_0, {B}))) := $s_0 \in$ B). Para los guardas que hacen referencia a las relaciones de tamaño, α_{guard} es equivalente a α_{size}.

Defínción (relación de coste). Supongamos que R_m es la representación recursiva de un método m donde cada bloque adopta la forma p($\bar{x}$) $\leftarrow G_p$, B, ($q_1(\bar{y})$; ...; $q_n(\bar{y})$) y $\hat{l_p}$ es la secuencia de variables que son relevantes. Sea ψ las relaciones de tamaño de R_m, donde cada relación de tamaño es de la forma $\langle$p($\bar{x}$), p'($\bar{x}$), $\psi^{p(z)}_{p'(z)}\rangle$ para todo p(z) $\in$ calls(B)$\bigcup$ {q(y)} donde q(y) se refiere al punto del programa inmediatamete después de B. A partir de esto, se generan las ecuaciones de coste para cada bloque como se muestra en el cuadro 3.13.

Donde T_b es el coste unidad asociado al bytecode b. La relación de coste asociado a R_m y ψ se define como el conjunto de ecuaciones de coste de sus bloques.

Hay que tener en cuenta cuatro puntos esenciales en la definición de arriba:

1. Las relaciones de tamaño entre las variables de entrada suministradas por el análisis de tamaño se unen a la ecuación de coste para p.

2. Los guardas no afectan al coste: sólo se utilizan para definir la aplicabilidad de las ecuaciones en función de las condiciones.

$$C_p(\hat{l_p}) \texttt{ = } \sum_{b \in bytecode(B)} T_b \texttt{ + } \sum_{r(z) \in calls(B)} C_r(\hat{l_r}) \texttt{ + } C_{pcont}(\bigcup_{i=1}^{n} \hat{l_{q_i}}) \bigwedge_{r(z) \in calls(B)} (\psi_{r(z)}^{p(x)} \wedge \psi_{q(y)}^{p(x)})$$

$$C_{pcont}(\bigcup_{i=1}^{n} \hat{l_{q_i}}) \texttt{ = } C_{q_1}(\hat{l_{q_1}}) \texttt{ si } \alpha_{guard}(G_{q_1})$$

$$\ldots$$

$$C_{pcont}(\bigcup_{i=1}^{n} \hat{l_{q_i}}) \texttt{ = } C_{q_n}(\hat{l_{q_n}}) \texttt{ si } \alpha_{guard}(G_{q_n})$$

Cuadro 3.13: Ecuaciones de coste

3. Los argumentos de las ecuaciones de coste son solo los argumentos relevantes del bloque. En la ecuación para la continuación, se necesita incluir la unión de todos los argumentos relevantes para cada uno de los bloques siguientes $Blocks_{q_i}$.

4. El coste T_b de una instrucción b depende modelo de coste que se elija. Si el interés es solo encontrar la complejidad o aproximar el número de bytecodes que serán ejecutados, entonces T_b puede ser 1 para todas las instrucciones. Por otra parte, se podrían usar modelos de coste más refinados para obtender el tiempo de ejecución de los métodos. Estos modelos pueden asignar diferentes costes en función de las instrucciones.

3.5.1. Ejemplo 1

Consideremos la representación recursiva del cuadro 3.2.1(sin las variables irrelevantes). Consideremos también las relaciones de tamaño del cuadro **??**. Aplicando la definición de relación de coste se obtienen las ecuaciones de coste del cuadro

T_{B_i} denota la suma del coste de todas las instrucciones de bytecode contenidas en el bloque $Block_1$.

$C_{add}(l_1, l_2) = C_{add^0}(l_1, l_2)$	
$C_{add^0}(l_1, l_2) = T_0 + C_{add^1}(l_1, l_2, l'_4)$	$l'_4 = 0$
$C_{add^1}(l_1, l_2, l_4) = T_1 + C_{add^1\_cont}(l_1, l_2, l_4)$	
$C_{add^1\_cont}(l_1, l_2, l_4) = C_{add^2()}$	$l_4 > l_1$
$C_{add^1\_cont}(l_1, l_2, l_4) = C_{add^3}(l_1, l_2, l_4)$	$l_4 \leq l_1$
$C_{add^2}() = T_2$	
$C_{add^3}(l_1, l_2, l_4) = T_3 + C_{add^3\_cont}(l_1, l_2, l_4)$	
$C_{add^3\_cont}(l_1, l_2, l_4) = C_{add^4}(l_1, l_2, l_4)$	$l_2 \in B$
$C_{add^3\_cont}(l_1, l_2, l_4) = C_{add^5}(l_1, l_2, l_4)$	$l_2 \in C$
$C_{add^3\_cont}(l_1, l_2, l_4) = C_{add^6}(l_1, l_2, l_4)$	$l_2 \in A$
$C_{add^4}(l_1, l_2, l_4) = T_4 + C_{B:incr}(l_2, l_4) + C_{add^8}(l_1, l_2, s_0)$	$s_0 = l_4 + 2$
$C_{add^4}(l_1, l_2, l_4) = T_5 + C_{C:incr}(l_2, l_4) + C_{add^8}(l_1, l_2, s_0)$	$s_0 = l_4 + 3$
$C_{add^4}(l_1, l_2, l_4) = T_6 + C_{A:incr}(l_2, l_4) + C_{add^8}(l_1, l_2, s_0)$	$s_0 = l_4 + 1$
$C_{add^8}(l_1, l_2, s_0) = T_8 + C_{add^1}(l_1, l_2, s_0)$	

Cuadro 3.14: Relaciones de tamaño del método de Fibonacci

3.5.2. Ejemplo 2

Consideremos las ecuaciones de coste para el método fibonacci en el cuadro 3.15.

Por ejemplo, la regla fib^a se utiliza para generar C_{fib}. Como fib contiene dos posibilidades en su cuerpo se genera una continuación. La continuación CC_0 tiene tantas alternativas como llamadas en las intersecciones, en este caso 2. Cada una de las dos se decora con su correspondiente guarda el cual asegura su aplicabilidad. Por lo tanto, la ecuación C_1(respectivamente C_2) solo se aplica si n es igual a 0(respectivamente diferente de 0). Las relaciones de tamaño también se consideran en cada una de las ecuaciones de coste que no correspondan a continuaciones. Por ejemplo, la ecuación C_4, que se asocia a la regla fib_4, hace uso de las dos relaciones de tamaño que relacionan fib_4 con fib_b y fib_c. La aplicación de tal relación de tamaño permite generar las correspondientes llamadas $C_{fib}(n-1)$ y $C_{fib}(n-2)$.

C_{fib}(n) = T_{Block0} + CC_0(n)

CC_0(n) = C_1(n) si $< n = 0 >$

CC_0(n) = C_2(n) si $< n \neq 0 >$

C_1(n) = T_{Block1} + C_5(n)

C_5(n) = T_{Block5}

C_2(n) = T_{Block2} + CC_2(n)

CC_2(n) = C_3(n) si $< n = 1 >$

CC_2(n) = C_4(n) si $< n \neq 1 >$

C_3(n) = T_{Block3} + C_5(n)

C_4(n) = T_{Block4} + C_{fib}(n - 1) + C_{fib}(n - 2)

Cuadro 3.15: Ecuaciones de coste del método de Fibonacci

Capítulo 4

Problemas de la representación y sus soluciones

4.1. Abstracción de bucles

EL grafo de control de flujo (CFG) es un componente esencial para llevar a cabo el análisis de los programas. Esencialmente, el CFG es una representación estática del bytecode que incorpora todos los flujos de control que pueden surgir durante la ejecución del bytecode.

Cuando uno está interesado en inferir propiedades a un nivel de verificación del bytecode, el uso del estándar CFG es suficiente para establecer unas ecuaciones sobre el bytecode. Una observación importante es que cuando se está interesado en inferir propiedades globales como terminación o cotas(superiores o inferiores) al coste, es esencial transformar el desestructurado flujo de control del bytecode en una forma estructurada apropiada. Por ejemplo, para inferir cotas superiores del coste de un programa, el análisis del coste crea relaciones de coste abstrayendo la estructura recursiva del programa e infiriendo relaciones de tamaño entre los argumentos. Es, por tanto, necesario representar el bytecode con una estructura recursiva para la cual las relaciones de coste puedan ser definidas. Para la terminación se necesita razonar sobre el comportamiento de terminación de diferentes bucles ocultos en el bytecode de bajo nivel que se puede originar a través de diferentes fuentes (por ejemplo, saltos condicionales e incondicionales, llamadas a métodos, o incluso excepciones). Es por tanto, otra vez necesario estructurar el bytecode en una forma apropiada gracias a la cual los bucles se puedan observar.

Estudios recientes para la terminación y el análisis del coste proponen repre-

sentar el CFG de una manera procedimental por medio de la (estructurada) representación recursiva (RR). Esta representación consiste en un conjunto de reglas con guardas que se obtienen directamente a partir de los bloques del CFG. Por lo tanto, la estructura de la RR refleja la forma del CFG del que fue obtenida. El objetivo final es que el bytecode se transforme en una estructurada RR donde todas las posibles formas de bucles del bytecode estén representadas de una manera uniforme por medio de la recursión. El análisis opera sobre las RR más bien que sobre el bytecode original (o sobre su CFG).

La observación principal de esta subsección es que la representación del CFG (y por lo tanto su correspondiente RR) tiene un impacto fundamental en las técnicas de análisis que buscan la inferencia de propiedades globales del programa.

Lo más importante es que, inspirándose en conocidas técnicas utilizadas en el campo de la descompilación, se propone contar con CFG's reducibles que contengan subgrafos independientes para cada bucle del programa en lugar de tener un único CFG en el que los bucles posiblemente estén conectados. La ventaja es que usando este tipo de CFG's se producen RR que mantienen separados los diferentes bucles y facilitan un razonamiento conjunto del bytecode. Como se verá más adelante, esto es esencial para inferir propiedades globales sobre el bytecode, como la terminación, que no se pueden inferir usando los CFG estándar.

4.1.1. Ejemplo

Consideremos el siguiente método estático sum(n, m) dentro del cuadro 4.1 que calcula $\sum_{i=0}^{n} \sum_{j=0}^{2m}$ i + j.

```
static int sum (int n, int m){
 int res = 0;
 for (int i = 0; i <= n; i++)
  for(int j = 0; j <= 2 * m; j++) res += i + j;
 return res;
}
```

Cuadro 4.1: código Java del método sum

El bytecode de sum(n, m) se muestra en el cuadro 4.2.

```
0: iconst_0       17: if_icmpgt 33
1: istore_2       20: iload_2
2: iconst_0       21: iload_3
3: istore_3       22: iload_4
4: iload_3        24: iadd
5: iload_0        25: iadd
6: if_icmpgt 39   26: istore_2
9: iconst_0       27: iinc 4,1
10: istore_4      30: goto 12
12: iload_4       33: iinc 3,1
14: iconst_2      36: goto 4
15: iload_1       39: iload_2
16: imult         40: ireturn
```

Cuadro 4.2: Bytecode de Java del método sum

Los índices 0..4 en el bytecode corresponden respectivamente a las variables locales n, m, res, i y j.

En la figura 4.1 se muestra su grafo de control de flujo.
En el CFG para `sum`, la bifurcación en el $Block_1$ (la cual contiene la condición del bucle externo) distingue entre el cuerpo del bucle ($Block_3$) y la salida del bucle ($Block_2$). El hecho de que el sucesor de la instrucción 6 (`if_icmpgt`

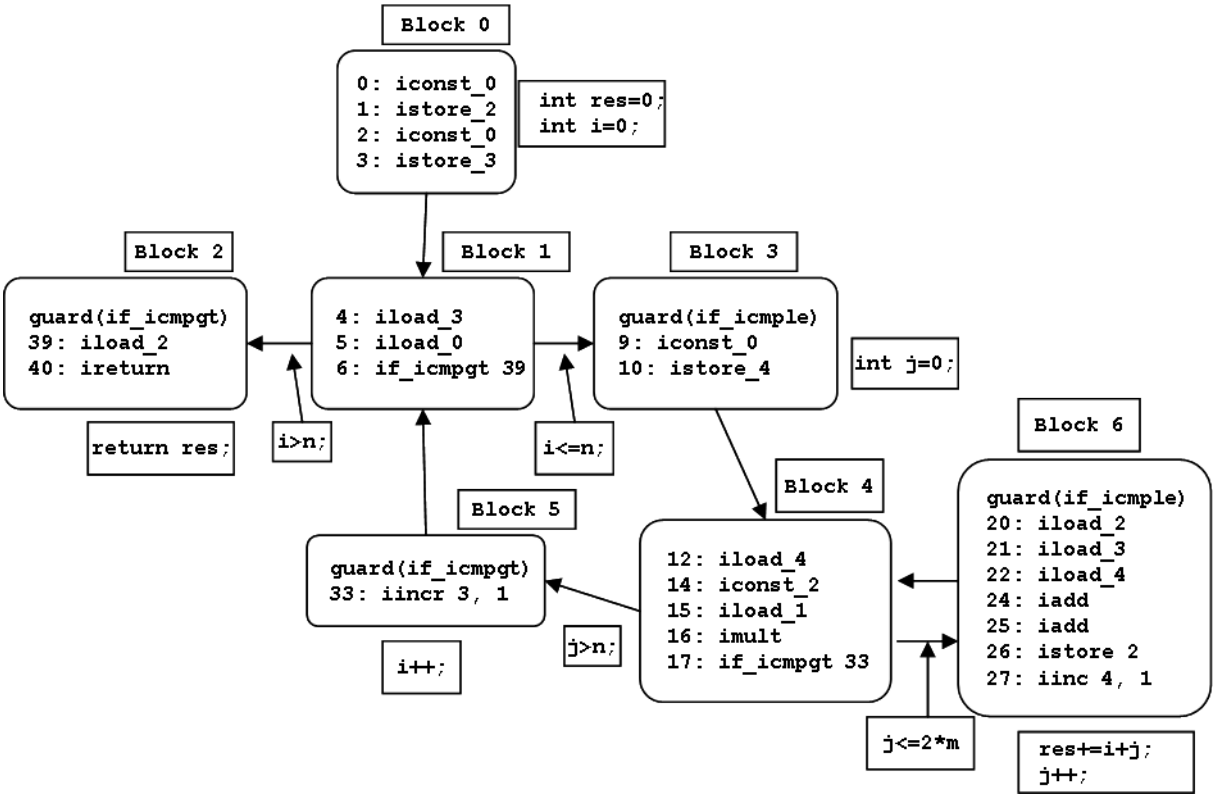


Figura 4.1: Grafo de control de flujo sin abstracción de bucles

39) pueda ser la instrucción de dirección 9 o la 39 se expresa a partir de dos arcos desde el $Block_1$ hasta $Block_2$ o hasta $Block_3$ y a partir de dos guardas: `guard(if_icmpgt)` del $Block_2$ y `guard(if_icmple)` del $Block_3$. El bucle interno se representa con los bloques 4, 5 y 6. El primero contiene la condición del bucle y los bloques $Block_5$ y $Block_6$ soportan la salida del bucle su cuerpo, respectivamente. Es muy importante observar que existe un arco desde el $Block_5$ hasta el $Block_1$, simulando el comportamiento de los dos bucles anidados. Se observa entonces que ambos bucles son dependientes, lo que en la RR nos llevaría tener recursión mutua.

En el cuadro 4.3 se muestra la RR del CFG de la figura 4.1, donde las reglas sum_1 y sum_4 hacen referencia al bucle externo e interno respectivamente, y donde se pone de manifiesto la recursión mutua, que lógicamente aparecerá en las ecuaciones de coste asociadas.
En este ejemplo, se obtienen las siguientes llamadas a las relaciones de tamaño. Se observa en el cuadro 4.4.

sum(n, m, ret) ← sum_0(n, m, res, i, j, ret).
sum_0(n, m, res, i, j, ret) ← **iconst**(0, s_0), **istore**(s_0, res), **iconst**(0, s_0), **istore**(s_0, i), sum_1(n, m, res, i, j, res).

sum_1(n, m, res, i, j, s_0, s_1, ret) ← **iload**(i, s_0), **iload**(n, s_1), (sum_2(n, m, res, i, j, s_0, s_1, ret); sum_3(n, m, res, i, j, s_0, s_1, ret)).

sum_2(n, m, res, i, j, s_0, s_1, ret) ← **guard**(bf if_icmpgt)s_0, s_1)), bf iload(res, s_0), bf ireturn(s_0, ret).

sum_3(n, m, ret, i, j, s_0, s_1, ret) ← **guard**(**if_icmple**(s_0, s_1)), **iconst**(0, s_0), **istore**(s_0, j), sum_4(n, m, res, i, j, ret).

sum_4(n, m, res, i, j, ret) ← **iload**(j, s_0), **iconst**(2, s_1), **iload**(m, s_2), **imul**(s_1, s_2, s_1), sum_5(n, m, res, i, j, s_0, s_1, ret); sum_6(n, m, res, i, j, ret)).

sum_5(n, m, res, i, j, s_0, s_1, ret) ← **guard**(**if_icmpgt**(s_0, s_1)), **iinc**(i, 1), sum_1(n, m, res, i, j, ret).

sum_6(n, m, res, i, j, s_0, s_1, ret) ← **guard**(**if_icmple**(s_0, s_1)), **iload**(res, s_0), **iload**(i, s_1), **iload**(j, s_2), **iadd**(s_1, s_2, s_1), **iadd**(s_0, s_1, s_0), **iinc**(j, 1), **istore**(s_0, res), sum_4(n, m, res, i, j, ret).

Cuadro 4.3: Representación intermedia del método sum

Grafos como el que se muestra en la figura 4.1 se convierten en inadecuados cuando se quiere tratar con algunos tipos de análisis estáticos del bytecode de Java, como el coste o la terminación. Concretamente, el problema real radica en la terminación, ya que la entrada al bucle interno se hace con j = 0 y en la salida j decrece. Los análisis de terminación existentes, y que se usan en nuestro análisis de coste, no saben determinar terminación en estos casos. La terminación de un programa es vital para asegurar que las ecuaciones de coste resultantes tienen solución. Para poder garantizarlo, los bucles del programa tienen que ser entidades independientes para poder razonar separadamente

sobre ellas.
En la figuras 4.2, 4.3 y 4.4 se muestra el grafo de control de flujo una vez realizada la abstracción de bucles.

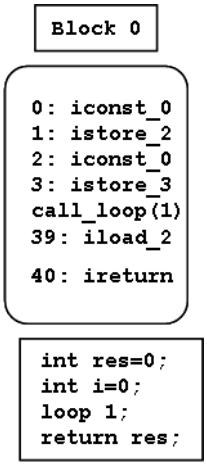


Figura 4.2: Grafo de control de flujo con abstracción de bucles. Primera parte.

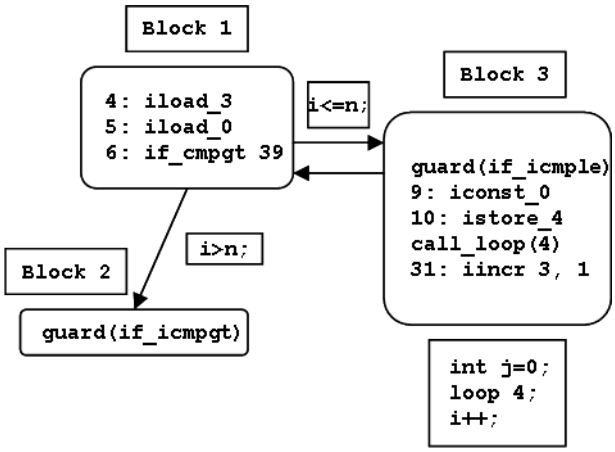


Figura 4.3: Grafo de control de flujo con abstracción de bucles. Segunda parte.

Son una modificación del CFG de la figura 4.1 en donde se han abstraído ambos bucles. Cada bucle se representa como un CFG independiente. Por lo tanto, el CFG de la figura 4.3 corresponde al bucle externo y contiene su cuerpo ($Block_3$) y una llamada al bucle interno (call_loop(4)). El bucle

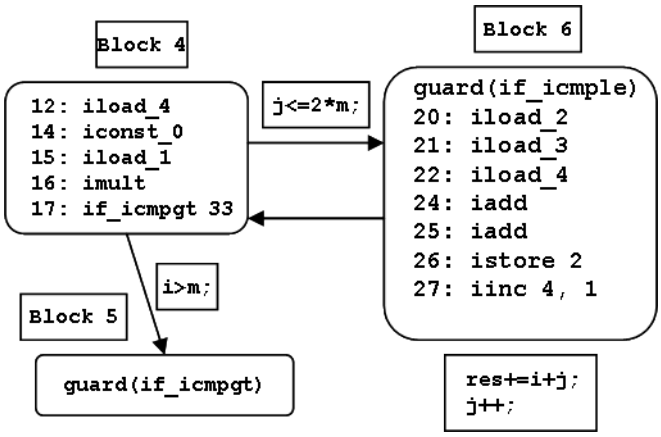


Figura 4.4: Grafo de control de flujo con abstracción de bucles. Tercera parte.

interno es el CFG de la figura 4.4. Nótese que este bucle no conecta nunca más con el externo como ocurría en la figura 4.1. Finalmente, el CFG principal, que encapsula la ejecución completa del programa, se muestra en la figura 4.2 y contiene una llamada al bucle externo `(call_loop(1))`. Mientras que se está simulando la ejecución de un programa una instrucción de la forma `call_loop(i)` otorga el control al CFG que comienza con el bloque $Block_i$(un bucle) y solo cuando se recorre este CFG el control vuelve al primero para proseguir la ejecución. Por medio de la abstracción de bucles, se puede analizar bucle de manera separada para calcular diversas propiedades y concretamente su terminación.
La representación intermedia para el grafo de las figuras 4.2, 4.3 y 4.4, después de hacer el proceso de 'slicing' es la que se muestra en el cuadro 4.5, donde se observa que la recursión mutua ha desaparecido
Ahora, se obtienen las relaciones de tamaño que se muestran el cuadro 4.6.

A continuación, en el cuadro **??** mostramos las ecuaciones de coste para `Sum`. Nótese que en estas ecuaciones se ha realizado lo que llamamos evaluación parcial. La evaluación parcial se explica en una sección posterior:

4.2. Ecuaciones de coste en forma canónica

Hasta ahora hemos presentado la forma general de las ecuaciones de coste que se obtienen a través del análisis de coste presentado en [8]. Estas ecuaciones tienen una forma muy genérica, ya que:

- Admiten varios argumentos en las llamadas (véase la RR para el método sum)
- Pueden contener varias ecuaciones recursivas.
- pueden se indeterministas debido a la falta de precisión cuando se realiza en análisis de tamaño.

La falta de precisión aparece, en general, cuando se manejan estructuras como arrays o listas, o cuando aparece el enlace dinámico [11]. Para el caso de arrays y listas, el problema es que se abstraen a su longitud, perdiéndose toda la información sobre sus elementos. Para el caso del enlace dinámico ocurre que guardas de la forma $o \in A$ (el objeto o pertenece a la clase A) no pueden traducirse a una relación de tamaño, lo que hace, como se observa en el ejemplo ...., que aparezcan tres ecuaciones que pueden ser aplicadas bajo las mismas condiciones.
Lógicamente, el objetivo de las ecuaciones de coste es poder ser resueltas para obtener la complejidad de un programa. Para el caso de nuestras ecuaciones, podriamos utilizar sistemas existentes como Mathematica ([21])para encontrar una forma cerrada, en concreto una solución exacta. Pero estos sistemas no son capaces, en algunos casos, de resolver nuestras ecuaciones, ni en general ecuaciones de coste inferidas de forma automática para programas reales [17] y [24]. Los motivos son varios:

1. Nuestras ecuaciones de coste pueden ser indeterministas, i.e., no existe una solución exacta. Por lo tanto los sistemas anteriores fallarían al intentar encontrarla.
2. Podemos tener varios argumentos. Esto supondría el que tener que hacer una transformación sobre las ecuaciones para conseguir pasarlas a un único argumento. Este proceso es muy tedioso y no siempre es alcanzable.
3. Cabe la posibilidad de que existan varias ecuaciones recursivas, lo que complica aún más el uso de sistemas algebraicos como los mencionados con anterioridad.

Dada la imposibilidad, en general, de obtener soluciones exactas, hemos decidido comprobar la complejidad de las ecuaciones analizando su forma sintáctica y apoyándonos es esquemas conocidos de complejidad. Por ejemplo, se sabe que una ecuación de recurrencia de la forma:

`T(0) = k1 si n` $\leq$ `0,`

```
T(n) = k2 + T(n-a), si n>0, a>1
```

tiene coste lineal. Para poder aplicar este tipo de esquemas a nuestro comprobador, aun tenemos que llevar a cabo ciertas transformaciones y comprobaciones.

Para empezar, y como se ha podido ver a lo largo de los ejemplos, nuestras ecuaciones no tienen recursión directa. Mediante la abstracción de bucles hemos conseguido eliminar la recursión mutua, sin embargo todavía las ecuaciones no responden a ningún esquema. Por ejemplo, en el método sum, una vez que hemos abstraido los bucles la recursión no es directa y necesitamos aplicar evaluación parcial por medio de desplegar las ecuaciones. Otro problema adicional, es que necesitamos garantizar que las ecuaciones son terminantes, ya que si no terminan no podrían decir cuál es su complejidad.

A continuación comentamos brevemente el proceso de transformación que sufrirán las ecuaciones de coste para hacerlas encajar con los esquemas.

4.2.1. Evaluación Parcial

Consiste básicamente en llevar a cabo un proceso de desplegado sobre las llamadas que no son recursivas. Lo mostramos a través de un ejemplo, concretamente es el método que calcula el factorial de un número y su código .java se muestra 4.8. el el cuadro

```
class Factorial {
   static int fact(int n){
     if (n == 0) return 1;
     else return n*fact(n-1);
   }
};
```

Cuadro 4.8: Código .java del factorial de un número n

A continuacion en el cuadro 4.9 mostramos las ecuaciones de coste sin aplicar evaluación parcial.

```
Factorial:fact(I)I[a] == m0[a],
    guard:
    size:
m1[] == 0,
    guard:
    size:
m0[a] == m2[a],
    guard:
    size:
m3[a] == 2 + m1[],
    guard:
    size: a=0
m4[a] == 7 + Factorial:fact(I)I[b] + m1[],
    guard:
    size: a-b=1 , a>= 1
m2[a] == 2 + m5[a],
    guard:
    size: m5[a] == m4[a],
    guard:a≠0
    size:
m5[a] == m3[a],
    guard:a=0
    size:
```

Cuadro 4.9: Ecuaciones de coste de factorial sin aplicar evaluación parcial

El complejidad del método que calcula el factorial de un número es lineal pero no se observa en el esquema mostrador en el cuadro 4.9 por eso hacemos la evaluación parcial que se observa en el cuadro 4.10.

```
Factorial:fact(I)I[a] == 9 + Factorial:fact(I)I[b],
    size: a>=1 , a-b=1
Factorial:fact(I)I[a] == 4,
    size: a=0
```

Cuadro 4.10: Ecuaciones de coste de factorial aplicando evaluación parcial

Una vez realizada la evaluación parcial se observa que las llamadas se han desplegado. Por ejemplo, vemos que la primera ecuación desplegada es `Factorial:fact(I)I[a] == m0[a] == m0[a] == m2[a] == 2 + m5[a] == m4[a] == 2 + 7 +`

`Factorial:fact(I)I[b] +` m_1`[] == 9 + Factorial:fact(I)I[b]`. Las relaciones de tamaño son a-b=1 y a>=1 ya que para que se aplica m_4[a] se tienen que dar estas condiciones. La segunda ecuación es `Factorial:fact(I)I[a] ==` m_0`[a] ==` m_0`[a] ==` m_2`[a] == 2 +` m_5`[a] ==` m_3`[a] == 2 + 2 +` m_1`[] == 4` y corresponde a otra posible bifurcación del flujo de control debido a que m_5[a] tiene dos opciones, en este caso la aplicaremos si a=0, por ello la relación de tamaño de esta ecuación es a=0.

Una vez hecho el proceso de evaluación parcial, las ecuaciones pasan a tener el siguiente formato:

Defínción. Relación de coste canónica Una relación de coste se dice canónica si todas las recursiones en el cuerpo de la regla son directas además de que todas las funciones son recursivas excepto la función principal

.

A estas ecuaciones las llamaremos ecuaciones en forma canónica, y son las que manejaremos a partir de ahora.

4.2.2. Terminación

Una vez que tenemos las ecuaciones en forma canónica, antes de comprobar su complejidad es necesario comprobar que son terminantes. Para garantizar la terminación de estas ecuaciones, se utilizan técnicas conocidas que consisten en garantizar que en todos los bucles existe algún parámetro (o combinación de ellos) que decrece. Por ejemplo en el método factorial en cada nueva llamada recursiva observamos que el argumento decrece en una unidad con respecto al argumento de la cabeza, además sabemos que termina pues el valor de argumento está acotado gracias al caso base. Esto es:

```
if (n==0) return 1;
else return n*fact(n-1);
```

En la segunda ecuación n decrece en 1 y en la primera n está acotado en 0 por lo tanto se puede garantizar que el método es terminante.

Como es sabido el problema de la terminación es indecidible en incluso hay casos en los que los programas serán terminantes y no seamos capaces de inferirlo. Para tales casos diremos que las ecuaciones de coste no están acotadas.

4.3. Conclusión

Lo más importante es que una vez realizadas todas estas transformaciones nuestras ecuaciones tienen las siguientes características:

- Tienen recursión directa.
- Pueden existir varios casos base.
- Pueden existir varias ecuaciones de recurrencia.
- Pueden tener varios argumentos.
- Pueden ser indeterministas.

Por todo esto y como ya hemos dicho antes, se nos presenta la imposibilidad de resolverlas en sistemas como Marthematica [21] y así hallar su complejidad. Así que tenemos que analizar la forma sintáctica de las ecuaciones, a continuación mostramos ya qué tipo de complejidades podemos manejar, y cómo realizamos el proceso de comprobación, todo ello en el capítulo 'Comprobación de complejidades.

$<$sum(n, m) $\mapsto$
sum_0(n', m', res, i, j), {n' = n, m = m'}$>$

$<$ sum_0(n, m, res, i, j) $\mapsto$
sum_1(n', m', res', i', j'),
{n' = n, m' = m, res' = 0, i' = 0, j' = j}$>$

$<$ sum_1(n, m, res, i, j) $\mapsto$
sum_2(n', m', res', i', j', s_0, s_1),
{n'= n, m' = m, res' = res, i' = i, j' = j, s_0 = i, s_1 = n,
$s_0 > s_1$}$>$

$<$ sum_1(n, m, res, i, j) $\mapsto$
sum_3(n', m', res', i', j', s_0, s_1),
{n' = m, m' = m, res' = res, i' = i, j' = j, s_0 = i, s_1 = n,
$s_0 <= s_1$}$>$

$<$ sum_3(n, m, res, i, j, s_0, s_1) $\mapsto$
sum_4(n', m', res', i', j'),
{n' = n, m' = m, res' = res, i' = i, j' = 0}$>$

$<$ sum_4(n, m, res, i, j) $\mapsto$
sum_5(n', m', res', i', j', s_0, s_1), {n' = n, m' = m, res' = res,
i' = i, j' = j, s_0' = j, s_1' = 2 * m, $s_0 <= s_1$)}$>$

$<$ sum_4(n, m, res, i, j) $\mapsto$
sum_6(n', m', res', i', j', s_0, s_1), {n' = n, m' = m, res' = res,
i' = i, j' = j, s_0' = j, s_1' = 2 * m, $s_0 > s_1$)}$>$

$<$ sum_5(n, m, res, i, j, s_0, s_1) $\mapsto$
sum_1(n', m', res', i', j'),
{n' = n, m' = m, res' = res, i' = i + 1, j' = j}$>$

$<$ sum_6(n, m, res, i, j, s_0, s_1) $\mapsto$
sum_4(n', m', res', i', j'),
{n' = n, m' = m, i' = i, j' = j + 1}$>$

Cuadro 4.4: Relaciones de tamaño del método sum

```
sum(n, m, ret) ← sum_0(n, m, res, i, j, ret).

sum_0(n, m, res, i, j, ret) ← iconst(0, s_0),
istore(s_0, res), iconst(0, s_0), istore(s_0, i),
sum_1(n, m, res, i, j, res), iload(res, s_0), ireturn(s_0, ret).

sum_1(n, m, res, i, j, ret) ← iload(i, s_0), iload(n, s_1),
(sum_2(n, m, res, i, j, s_0, s_1, ret);
(sum_3(n, m, res, i, j, s_0, s_1, ret)).

sum_2(n, m, res, i, j, s_0, s_1, ret) ←
guard(if_icmpgt(s_0, s_1).

sum_3(n, m, res, i, j, s_0, s_1, ret) ←
guard(if_icmple(s_0, s_1),
iconst(0, s_0), istore(s_0, j),
sum_4(n, m, res, i, j, ret), iinc(i, 1),
sum_1(n, m, res, i, j, ret).

sum_4(n, m, res, i, j, ret) ← iload(j, s_0),
imul(s_1, s_2, s_1),
(sum_5(n, m, res, i, j, s_0, s_1, ret);
sum_6(n, m, res, i, j, s_0, s_1, ret)).

sum_5(n, m, res, i, j, s_0, s_1, ret) ←
guard(if_cmpgt(s_0, s_1)).

sum_6(n, m, res, i, j, s_0, s_1, ret) ←
guard(if_cmple(s_0, s_1)),
iload(res, s_0), iload(i, s_1), iload(j, s_2), iadd(s_1, s_2, s_1)
iadd(s_0, s_1, s_0), istore(res, s_0), iinc(j, 1)
sum_4(n, m, res, i, j, ret).
```

Cuadro 4.5: Representación recursiva del método sum con abstracción de bucles

$<$sum(n, m) $\mapsto$ sum_0(n', m', res, i, j), {n' = n, m = m'}$>$

$<$ sum_0(n, m, res, i, j) $\mapsto$ sum_1(n', m', res', i', j'),
{n' = n, m' = m, res' = 0, i' = 0, j' = j}$>$

$<$ sum_1(n, m, res, i, j) $\mapsto$ sum_2(n', m', res', i', j', s_0, s_1),
{n'= n, m' = m, res' = res, i' = i, j' = j, s_0 = i, s_1 = n,
$s_0 > s_1$}$>$

$<$ sum_1(n, m, res, i, j) $\mapsto$ sum_3(n', m', res', i', j', s_0, s_1),
{n' = m, m' = m, res' = res, i' = i, j' = j, s_0 = i, s_1 = n,
$s_0 <= s_1$}$>$

$<$ sum_3(n, m, res, i, j, s_0, s_1) $\mapsto$ sum_4(n', m', res', i', j'),
{n' = n, m' = m, res' = res, i' = i, j' = 0}$>$

$<$ sum_3(n, m, res, i, j, s_0, s_1) $\mapsto$ sum_1(n', m', res', i', j'),
{n' = n, m' = m, i' = i' + 1, j' = j}$>$

$<$ sum_4(n, m, res, i, j) $\mapsto$ sum_5(n', m', res', i', j', s_0, s_1),
{n' = n, m' = m, res' = res, i' = i, j' = j, s_0' = j,
s_1' = 2 * m, $s_0 > s_1$)}$>$

$<$ sum_4(n, m, res, i, j) $\mapsto$ sum_6(n', m', res', i', j', s_0, s_1),
{n' = n, m' = m, res' = res, i' = i,
j' = j, s_0' = j, s_1' = 2 * m, $s_0 <= s_1$)}$>$

$<$ sum_6(n, m, res, i, j, s_0, s_1) $\mapsto$ sum_4(n', m', res', i', j'),
{n' = n, m' = m, i' = i, j' = j + 1}$>$

Cuadro 4.6: Relaciones de tamaño del método sum con abstracción de bucles

Sum:sum(II)I[a,b] == 6 + m_2[a,b,0],
size: b>=0 , a>=0
m_2[a,b,c] == 7 + m_0[b,c] + m_2[a,b,d],
size: a-c>=0 , b>=0 , c-d= -1
m_2[a,b,c] == 3,
size: b>=0
m_0[a,b] == 11 + m_0[a,c],
size: a-c>=-1 , b-c = -1
m_0[a,b] == 3,
size:

Cuadro 4.7: Ecuaciones de coste del método sum

Capítulo 5

Comprobación de complejidades

5.1. Introducción

Las complejidades que chequea nuestro sistema son:

- Complejidades constantes.
- Complejidades polinómicas de grado k.
- Complejidades exponenciales de base b.

No consideramos el orden logarítmico y los problemas 'divide y vencerás', porque requieren un conocimiento demasiado detallado de cómo decrecen los argumentos. Por ejemplo, para el orden logarítmico tendríamos que garantizar que en cada llamada recursiva el argumento (o combinación de argumentos) decrece en al menos 1/2. Este proceso es altamente complicado y queda fuera de los objetivos de este trabajo. Así pues, el coste logarítmico quedará encuadrado dentro del coste lineal, y por motivos similares, el coste n log n quedará acotado por el coste exponencial.
Normalmente los distintos esquemas de complejidades que estudiamos presentan sólo un argumento. En nuestro caso, las ecuaciones que estamos considerando poseen varios argumentos. Esto implica que debemos realizar las consideraciones que se realizan sobre un solo argumento en varios. Además de esto, debemos tener en cuenta que podemos tener varias ecuaciones de complejidad para el mismo procedimiento, es decir, debemos considerar cada ecuación por separado. Todas las variables del cuerpo de una ecuación son

combinación lineal de las variables que aparecen en la cabeza, para poder hacer la comprobación de complejidad. Es importante recalcar que está parte la estamos asumiendo pues no está implementada.

Respecto a los argumentos, consideramos sólo las complejidades que implican un decremento en los argumentos, es decir, en las complejidades polinómicas y exponenciales, las llamadas recursivas se hacen con argumentos que decrecen cosa que no ocurre, por ejemplo, en el caso de las complejidades logarítmicas.

El sistema además de chequear la complejidad, da una información más precisa. Es decir, si queremos saber que algo es cuadrático, entonces si lo es diremos que sí, pero si además es lineal informaremos sobre que es lineal.

5.2. Esquemas utilizados

El chequeo de la complejidad se basa en la utilización de las siguientes ecuaciones.

5.2.1. Complejidad constante

El esquema conocido de la complejidad constante es el que se muestra en el cuadro 5.1.

O(1)	T(n) = c, $\forall$c $\in \mathbb{Z}$

Cuadro 5.1: Complejidad constante

En nuestro caso al tener varios argumentos tendriamos lo que se muestra en el cuadro 5.2.

O(1)	$T(n_1, ..., n_n)$ = c, $\forall$c $\in \mathbb{Z}$

Cuadro 5.2: Complejidad constante

Las posibilidades que nos podemos encontrar son:

- $T(n_1, ..., n_n)$ = c

- $T(n_1, ..., n_n)$ = $C_0(x_1, ..., x_n) + ... + C_n(x_1, ..., x_n)$

Es decir, podemos, bien no encontrarnos ninguna llamada a procedimientos, por lo cual el coste es constante, o bien, si nos encontramos llamadas a otros procedimientos el coste de esto debe ser constante, es decir, el coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ deben tener coste constante.

5.2.2. Complejidad polinómica de grado M, siendo M > 0

El esquema conocido de la complejidad polinómica es el que se muestra en el cuadro 5.3.

O(n)	T(n) = a + T(n-b), n>0, a>0, b>0
	T(n) = 0, n≤0

Cuadro 5.3: Complejidad polinómica

En nuestro caso al tener varios argumentos tendríamos el esquema que se muestra en el cuadro 5.4.

O(n)	$T(n_1, ..., n_n)$ = a + T($n_1 - b$, ...,$n_n - b'$), n>0, a>0, b>0, b'>0
	$T(n_1, ..., n_n)$ = 0, $n_1 \leq 0$, ..., $n_n \leq 0$

Cuadro 5.4: Complejidad polinómica con varios argumentos

Ahora veamos las diferentes posibilidades que nos podemos encontrar:

- $T(n_1, ..., n_n)$ = c

- $T(n_1, ..., n_n)$ = $C_0(x_1, ..., x_n) + ... + C_n(x1', ..., x'_n)$

- $T(n_1, ..., n_n)$ = $C_0(x_1, ..., x_n) + ... + C_n(x1', ..., x'_n)$ + $C(n'_1, ..., n'_n)$

En el primer caso nos podemos encontrar con una complejidad constante. Esto se debe a que, al ser menor la complejidad constante que la complejidad

polinómica (puesto que estamos considerando una complejidad polinómica de grado > 0), la complejidad polinómica abarca la constante, es decir, si tenemos una complejidad constante, es una complejidad polinómica.

En el segundo caso no nos encontramos ninguna llamada recursiva. Es por eso, que la complejidad de los procedimientos invocados deben tener coste constante, polinómico de grado menor que m o polinómico de grado m. Hay que tener en cuenta, que si nos encontramos una llamada a un procedimiento con coste polinómico de grado m ya no nos podremos encontrar ninguna llamada recursiva. Este punto ha sido muy importante en la implementación. Es decir, el coste de $C_0(x_1, ..., x_n), ..., C_n(x'_1, ..., x'_n)$ debe ser polinómico de grado menor que m o bien, coste constante.

Pasamos a analizar el último punto. En este caso nos encontramos una llamada recursiva. Es por esto que las llamadas a procedimientos que se realicen en este caso deben tener coste constante o polinómico de grado menor que m. Si nos encontráramos un procedimiento con coste polinómico de grado m, al tener la llamada recursiva, el coste total del procedimiento sería de grado m+1. Es decir, el coste de $C_0(x_1, ..., x_n), ..., C_n(x'_1, ..., x'_n)$ debe ser constante o polinómico de grado menor que m.

A continuación vamos a ejemplificar lo explicado en este apartado con los ejemplos polinómicos típicos, como son los costes lineales, cuadráticos y cúbicos.

Polinómico de grado 1 (lineal)

$\star$ En el caso lineal puede ocurrir:

(1) $T(n_1, ..., n_n)$ = c (coste constante, al ser O(1)<O(n), O(1) $\in$ O(n)).

(2) $T(n_1, ..., n_n)$ = c + $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$
(a) El coste de $C_0(x_1, ..., x_n), \ldots, C_n(x'_1, ..., x'_n)$ puede ser lineal.
(b) El coste de $C_0(x_1, ..., x_n), \ldots, C_n(x'_1, ..., x'_n)$ puede ser constante.

(3) $T(n_1, ..., n_n) = C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n) + C(n'_1, ..., n'_n)$
(a) El coste de $C_0(x_1, ..., x_n), \ldots, C_n(x'_1, ..., x'_n)$ debe ser constante.

Polinómico de grado 2 (cuadrático)

⋆ En el caso cuadrático puede ocurrir:

(1) $T(n_1, n_n)$ = c (coste constante, al ser O(1)<O(n^2), O(1) $\in$ O(n^2)).

(2) $T(n_1, ..., n_n)$ = c + $C_0(x_1, \ldots, x_n)$ + ... + $C_n(x'_1, \ldots, x'_n)$
(a) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser cuadrático.
(b) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser lineal.
(c) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser constante.

(3) $T(n_1, ..., n_n)$ = $C_0(x_1, \ldots, x_n)$ + ... + $C_n(x'_1, \ldots, x'_n)$ + $C(n'_1, ..., n'_n)$
(a) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser constante.
(b) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser lineal.

Polinómico de grado 3 (cúbico)

⋆ En el caso cúbico puede ocurrir:

(1) $T(n_1, ..., n_n)$ = c (coste constante, al ser O(1)<O(n^3), O(1) $\in$ O(n^3)).

(2) $T(n_1, ..., n_n)$ = c + $C_0(x_1, \ldots, x_n)$ + ... + $C_n(x'_1, \ldots, x'_n)$
(a) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser cúbico.
(b) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser cuadrático.
(c) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser lineal.
(d) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser constante.

(3) $T(n_1, ..., n_n)$ = $C_0(x_1)$ + ... + $C_n(x_n)$ + $C(n'_1, ..., n'_n)$
(a) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser constante.
(b) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser lineal.
(c) El coste de $C_0(x_1, ..., x_n)$, ..., $C_n(x'_1, ..., x'_n)$ puede ser cuadrático.

5.2.3. Complejidad exponencial de base B, siendo B > 1

El esquema conocido de la complejidad exponencial es el que se muestra en el cuadro 5.5.

$O(b^n)$	`T(n) = a + b*T(n-c), n>0, a>0, b>1, c>0`
	`T(n) = 0, n` $\leq$ `0`

Cuadro 5.5: Complejidad exponencial de base b

En nuestro caso al tener varios argumentos tendríamos el esquema que se muestra en el cuadro 5.6.

$O(b^n)$	$T(n_1, ..., n_n)$ `= a + b*T(`n_1`-c, ..., `n_n`-c), n>0, a>0, b>1, c>0, c'>0`
	$T(n_1, ..., n_n)$ `= 0, n` $\leq$ `0`

Cuadro 5.6: Complejidad exponencial de base b con varios argumentos

Ahora veamos las diferentes posibilidades que nos podemos encontrar:

- $T(n_1, ..., n_n)$ `= c`
- $T(n_1, ..., n_n)$ `=` $C_0(x_1, ..., x_n)$ `+ ... +` $C_n(x'_1, ..., x'_n)$
- $T(n_1, ..., n_n)$ `=` $C_0(x_1, ..., x_n)$ `+ ... +` $C_n(x'_1, ..., x'_n)$ `+` $C(n'_1, ..., n'_n)$
- $T(n_1, ..., n_n)$ `=` $C_0(x_1, ..., x_n)$ `+ ... +` $C_n(x'_1, ..., x'_n)$ `+` $b*C(n'_1, ..., n'_n)$

En el primer caso nos podemos encontrar con una complejidad constante. Esto se debe a que, al ser menor la complejidad constante que la complejidad exponencial, la complejidad exponencial abarca la constante, es decir, si tenemos una complejidad constante, es una complejidad exponencial.

En el segundo caso no nos encontramos ninguna llamada recursiva. Es por eso, que la complejidad de los procedimientos invocados deben tener coste constante, polinómico de cualquier grado o exponencial de base b o menor.

En el tercer punto nos encontramos una llamada recursiva. Es por esto que las llamadas a procedimientos que se realicen en este caso deben tener coste constante o polinómico. Es decir, el coste de $C_0(x_1, ..., x_n), ..., C_n(x'_1, ..., x'_n)$ debe ser constante o polinómico.

En el último caso tenemos b llamadas recursivas (recordamos que b es la base de la complejidad exponencial a chequear). Esto implica que las llamadas al resto de procedimientos deben tener coste constante debido a la definicón.

A continuación vamos a ejemplificar lo explicado en este apartado con los ejemplos exponenciales típicos, como son los costes exponenciales de base 2 y exponenciales de base 3.

Exponencial de base 2

⋆ En el caso exponencial de base 2 puede ocurrir:

(1) $T(n_1, ..., n_n)$ = c (coste constante, al ser O(1)<O(2^n), O(1) $\in$ O(2^n)).

(2) $T(n_1, ..., n_n)$ = c + $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$
(a) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser exponencial base 2.
(b) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser cúbico.
(c) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser cuadrático.
(d) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser lineal.
(e) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser constante.

(3) $T(n_1, ..., n_n)$ = c + $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n) + C(n'_1, ..., n'_n)$
(a) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser constante.
(b) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser lineal.
(c) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser cuadrático.
(d) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser cúbico.

(4) $T(n_1, ..., n_n) = C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n) + C(n'_1, ..., n'_n) + C(n''_1, ..., n''_n)$
(a) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ debe ser constante.

Exponencial de base 3

⋆ En el caso exponencial de base 3 puede ocurrir:

(1) $T(n_1, ..., n_n)$ = c (coste constante, al ser O(1)<O(3^n) O(1) $\in$ O(3^n)).

(2) $T(n_1, ..., n_n)$ = c + $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$
(a) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser exponencial base 3.
(b) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser exponencial base 2.
(c) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser cúbico.
(d) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser cuadrático.
(e) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser lineal.
(f) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser constante.

(3) $T(n_1, ..., n_n) = C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n) + C(n'_1, ..., n'_n)$
(a) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser exponencial base 2.
(b) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser constante.
(c) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser lineal.
(d) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser cuadrático.
(e) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser cúbico.

(4) $T(n_1, ..., n_n) = C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n) + C(n'_1, ..., n'_n) + C(n''_1, ..., n''_n)$
(a) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser constante.

(5) $T(n_1, ..., n_n) = C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n) + C(n'_1, ..., n'_n) + C(n''_1, ..., n''_n) + C(n'''_1, ..., n'''_n)$

(a) El coste de $C_0(x_1, \ldots, x_n) + \ldots + C_n(x'_1, \ldots, x'_n)$ puede ser constante.

Capítulo 6

Descripción del sistema

6.1. Implementación del chequeo de la complejidad

Para poder realizar los cálculos necesarios para hallar la complejidad usamos un predicado que nos viene dado por la implementación inicial sobre la que estamos trabajando. Este predicado se denomina jcost y tiene cinco atributos. El primero de ellos es la cabecera del método que estamos analizando, que consta del nombre del método y de los atributos del mismo. El siguiente atributo es una lista con tres componentes. La primera de ellas es un número entero no relevante para el cálculo de la complejidad. La segunda es una lista formada por las llamadas que realiza el procedimiento considerado a otros métodos. La tercera componente es la llamada del método a otro bloque. El tercer atributo de los jcost es una lista de listas. Sus elementos son las ecuaciones de size que debemos tener en cuenta para ver la relación entre los argumentos del procedimiento tratado. Esta parte no ha sido implementada, es decir, consideramos que la relación entre los argumentos siempre va a ser lineal, para que se pueda hallar la complejidad polinómica o exponencial. El resto de los atributos de los jcost no son relevantes para esta parte de la implementación.

Existen dos implementaciones del cálculo de la complejidad. La primera de ellas es sin parametrizar, es decir, existe un predicado que comprueba si un método es constante, lineal, cúbico cuadrático, exponencial de base 2 o exponencial de base 3. La segunda es parametrizada, es decir, poseemos dos predicados, polinómico y exponencial, y a cada predicado le mandamos un entero que en el caso el polinómico representa el grado del polinomio y en el caso del exponencial, la base. Pasamos a explicar las dos implementaciones.

6.1.1. Chequeo de la complejidad sin parametrización

En este caso, y como se ha explicado anteriormente, tenemos un método para cada complejidad a comprobar. Para cada uno de ellos, primeramente se buscan todos los jcost de la base de datos relacionados con el procedimiento que estamos considerando. Para cada jcost introducimos el nombre del método, las llamadas a procedimientos (el segundo parámetro de los jcost completo y luego ignoramos el primer atributo) y las ecuaciones de size en una lista. Este último atributo no lo vamos a tener en cuenta, puesto que serviría para comprobar la combinación lineal de atributos que le pasamos al procedimiento y los atributos con los que llamamos a otros procedimientos en la implementación. Una vez tenemos todo en una lista, para cada tupla obtenemos los atributos de la cabeza y los del cuerpo, así como las llamadas a otros procedimientos. Una vez realizado esto, la implementación de cada procedimiento es diferente según se ha explicado en el capítulo 5. Debemos tener en cuenta que vamos a analizar cada jcost uno a uno, es decir, analizamos cada tupla de la lista una a una. Para el caso del coste lineal, debemos comprobar que el coste de los métodos llamados por el procedimiento en cada jcost son, todos constantes o uno y sólo uno de los métodos del segundo atributo de los jcost puede coincidir con el nombre de la cabeza (primer atributo) de ese mismo jcost. Esto impica que tenemos una llamada recursiva y que el 'coste' de ese jcost es lineal. ¿Qué ocurriría si en lugar de una llamada recursiva tuvieramos dos? Que ya no sería lineal y sería exponencial de base 2. Puede ocurrir que en ese jcost no tengamos una llamada recursiva, pero tengamos una llamada a un procedimiento que tiene coste lineal. ¿Qué ocurriría si seguidamente encontráramos una llamada recursiva? Pues que el coste ya no sería lineal, sino que sería cuadrático.

Simplificando, si nos encontramos una llamada a procedimiento recursiva, entonces el resto de los procedimientos deben tener coste constante. Si por el contrario nos encontramos una llamada a un procedimiento de coste lineal, entonces el resto de los procedimientos llamados por el jcost que estamos considerando deben tener coste constante o coste lineal.

En este caso, el coste que pueden tener los jcost son constante o lineal, puesto que como es sabido, la suma de costes constantes da como resultado un coste constante (que al ser un coste inferior al lineal, si nos devuelve coste constante podremos contestar cierto pero especificamos que es constante) y la suma de costes lineales da como resultado un coste lineal.

En el coste cuadrático tenemos más posibilidades para cada jcost. Puede ocurrir que todos los métodos invocados desde un jcost tengan coste constante. Puede ocurrir también que algunos tengan coste constante y otros tengan coste lineal. Especial atención debemos poner en los casos en que nos encontremos una llamada a un procedimiento de coste cuadrático. En este caso, ya no podremos encontrarnos ninguna llamada recursiva. Por el contrario, si primeramente nos encontramos una llamada recursiva, ya no nos podremos encontrar ni otra llamada recursiva(pues tendríamos coste exponencial) ni un procedimiento con coste cuadrático(pues tendríamos coste cúbico). Así pues, si nos encontramos con procedimientos de coste constante, el resto de llamadas del procedimiento pueden ser una llamada recursiva, un procedimiento con coste lineal o un procedimiento con coste cuadrático. Si nos encontramos un procedimiento con coste lineal, nos podremos seguir encontrando procedimientos con coste constante, lineal, cuadrático o una llamada recursiva. Si nos encontramos un procedimiento con coste cuadrático, debemos tener en cuenta que ya no nos podremos encontrar ninguna llamada recursiva. Si por el contrario nos encontramos con una llamada recursiva sólo nos podremos encontrar procedimientos con coste constante o lineal, no nos podremos encontrar otra llamada recursiva (coste exponencial) ni un coste cuadrático (coste cúbico).

Analizando todos los jcost juntos estos pueden tener, coste constante, lineal o cuadrático, puesto que, a saber, la suma de procedimientos de coste cuadrático da como resultado orden cuadrático.

El análisis de orden cúbico es similar a los anteriores. En este caso los jcost pueden tener coste constante, lineal, cuadrático o cúbico. Para cada jcost ahora debemos tener en cuenta que si alguna de las llamadas que se realizan en ese jcost tiene coste cúbico a continuación sólo podremos encontrarnos procedimientos con coste lineal, cuadrático o cúbico, es decir, no nos podremos encontrar ninguna llamada recursiva. Si nos encontramos una llamada recursiva, debemos tener en cuenta que sólo nos podemos encontrar procedimientos con coste constante, lineal o cuadrático, es decir, no nos podremos encontrar otra llamada recursiva ni un procedimiento de coste cúbico. Si por el contrario nos encontramos con procedimientos de coste constante, lineal o cuadrático, el resto de procedimientos invocados pueden tener coste constante, lineal, cuadrático, cúbico, o bien nos podremos encontrar una llamada recursiva (y sólo una según lo explicado anteriormente)

A continuación explicamos lo descrito anteriormente con un ejemplo. El pro-

cedimiento que se ha analizado en este caso es el cálculo del factorial de un número, y las ecuaciones obtenidas son (se muestran sólo las llamadas a procedimientos, es decir, no se muestran las ecuaciones de size ni los guardas):

```
Factorial:fact(I)I[a] == 9 + Factorial:fact(I)I[b],

Factorial:fact(I)I[a] == 4,
```

De todos es sabido que el coste de este procedimiento es lineal. Vamos a seguir los pasos que sigue nuestro predicado para comprobar que el coste es lineal.

Una de las llamadas del factorial tiene coste constante, pero la segunda posee una llamada recursiva, esto implica que el coste del procedimiento Factorial por lo especificado anteriormente posee coste lineal.

Pasamos ahora a explicar como comprobamos los ordenes exponenciales de base dos y tres (recordamos que estamos calculando los órdenes sin parametrizar).

Para los dos casos comprobamos primeramente si el coste del procedimiento es menor que exponencial ,es decir, para el caso de exponencial de base 2 comprobamos si el coste es constante, lineal cuadrático o cúbico, y para el caso del exponencial de base 3 comprobamos si el coste del procedimiento es constante, lineal, cuadrático, cúbico o exponencial de base2.

Si el paso anterior no da éxito, entonces empezamos a comprobar el coste del procedimiento. Debemos recordar que un procedimiento es exponencial de base 2 si posee dos llamadas recursivas, y un procedimiento es exponencial de base 3 si posee tres llamadas recursivas, así que lo que hacemos es ir analizando los jcost. Nos centramos primero en el caso del exponencial de base 2. Si primeramente nos encontramos una llamada a un procedimiento de coste lineal, cuadrático, cúbico o exponencial de base 2, entonces ya no nos podremos encontrar ninguna llamada recursiva, es decir, nos podremos encontrar procedimientos de coste constante, lineal, cuadrático, cúbico o exponencial de base 2, pero no una llamada recursiva. Si al contrario nos encontramos coste constante podemos encontrarnos procedimientos de coste igual a alguno de los mencionados anteriormente, o bien, una llamada recursiva. Si nos encontramos una llamada recursiva, entonces no podremos encontrarnos ningún procedimiento de coste lineal, ni cuadrático, ni cúbico ni exponencial de base 2, pero si nos podremos encontrar otra llamada recursiva. Si nos encontramos esta llamada recursiva, entonces el resto de procedimientos deben tener coste

constante.

Similar a lo anterior es el caso de exponencial de base tres, pero, cuando nos encontremos la llamada recursiva, todavía nos podremos encontrar, o bien dos llamadas recursivas más, o bien procedimientos de coste constante.

Pasemos a analizarlo con un ejemplo, el problema del Hanoi, que como sabemos, tiene coste exponencial de base 2.

```
Hanoi:hanoi(IIII)V[a] == 3,,

Hanoi:hanoi(IIII)V[a] == 17 + Hanoi:hanoi(IIII)V[b]
+ Hanoi:hanoi(IIII)V[c],
```

Comprobamos que en este caso, una de las llamadas del Hanoi tiene coste constante, pero la segunda posee dos llamadas recursivas, así que el procedimiento tiene coste exponencial de base 2.

6.1.2. Chequeo de la complejidad con parametrización

En este caso vamos a tener dos predicados, parametrizado que comprueba si un cierto programa tiene coste polinómico de cualquier exponente entero, y parametrizadoExponential que comprueba si un cierto programa tiene coste exponencial de cualquier base.

Comenzamos con el predicado parametrizado. Primeramente comprobamos si es polinómico de grado menor que k, siendo k el valor que le pasamos al predicado como parámetro. Cuando llegamos a k=0 comprobamos si el coste del predicado es constante. De no ser polinómico de grado menor que k, entonces comprobamos si es polinómico de grado k. Primeramente debemos comprobar que en cada jcost sólo exista una llamada recursiva, puesto que si existe más de una llamada no sería coste polinómico sino que estaríamos analizando un coste exponencial. Una vez comprobado ese paso comprobamos si en el jcost analizado existe una llamada recursiva. Este paso puede parecer redundante, pero la comprobación previa nos indica sólo si existe alguna llamada recursiva, es decir, tiene éxito si existe una o ninguna. Si existe una llamada recursiva entonces el resto de predicados deben ser polinómicos de grado k-1. Si, por el contrario, no existe ninguna llamada recursiva en ese jcost, entonces el coste de los procedimientos a los que llama, deben ser polinómicos de grado

k.

Pasamos ahora a analizar el predicado parametrizadoExponential. En este caso primero debemos comprobar si el predicado tiene coste polinómico. Aunque es muy difícil de encontrar, calculamos si es polinómico de grado 10, de esta formas consideramos los ordenes polinómicos normales que son constante, lineal, cuadrático y cúbico. De no tener el procedimiento considerado coste polinómico comprobamos si es exponencial de base k, siendo k un entero pasado al predicado. Este análisis es más sencillo que el anterior. Lo que hacemos es, si nos encontramos una llamada recursiva, comprobamos que el resto de procedimientos deben tener coste exponencial de grado k-1, y si no nos encontramos ninguna llamada recursiva, entonces su coste debe ser constante y el coste del resto de procedimientos debe ser exponencial de grado k.

6.2. Implementación de la interfaz web del sistema

6.2.1. Conocimientos previos.

En los comienzos del desarrollo Web, las páginas Web mostraban información que no solía cambiar. Esta forma estática de mostrar información era bastante eficiente puesto que la página se creaba una única vez y se presentaba. Si era necesario se hacían mínimos cambios y ya estaba lista otra vez. Esta forma de desarrollo de páginas Web se quedó corta ya que surgió la necesidad de interactuar con el usuario y de adaptar la información a sus necesidades, o mostrar información que se toma de bases de datos que cambian frecuentemente. Por ello, aparecieron las técnicas de generación dinámica de páginas que permiten de forma relativamente fácil mantener actualizadas las páginas aunque se muestre información que cambia frecuentemente y también posibilitan formas de establecer comunicaciones personalizadas con los usuarios.

6.2.2. Tecnologías usadas para la creación de páginas dinámicas.

- CGI. Las siglas se corresponden con Common Gateway Interface (CGI): los primeros servidores HTTP no incluían ningún mecanismo para generar respuestas dinámicamente, por lo tanto se crearon interfaces para comunicar el servidor con programas externos que implementaran dicha funcionalidad.

- JAVA SERVLETS. Java Servlets: fueron introducidos por Sun en 1996 como pequeñas aplicaciones Java para añadir funcionalidad dinámica a los servidores Web. Los servlets, al igual que los scripts CGI, reciben una petición del cliente y generan los contenidos apropiados para su respuesta, aunque el esquema de funcionamiento es diferente.

- JSP. JavaServer Pages (JSP): es una tecnología híbrida basada en template systems. Puede incorporar scripts para añadir código Java directamente a las páginas .jsp, pero también implementa un conjunto de etiquetas que interaccionan con los objetos Java del servidor, sin la necesidad de que aparezca código fuente en la página.

Características y funcionamiento de alguna de las tecnologías.

1. SERVLETS.

   a) Características.
   Las características principales de los servlets entre otras son las siguientes:

   - Son independientes del servidor utilizado y de su sistema operativo. Gracias a esto, a pesar de que los Servlets estén escritos en Java, el servidor puede estar escrito en cualquier lenguaje de programación.

   - Los servlets pueden llamar a otros servlets, e incluso a métodos concretos de otros servlets (en la misma máquina o en una máquina remota). De esta forma se puede distribuir de forma más eficiente el trabajo a realizar. Por ejemplo, se podría tener un servlet encargado de la interacción con los clientes y que llamara a otro servlet para que a su vez se encargara de la comunicación con una base de datos.

- Los servlets pueden obtener fácilmente información acerca del cliente (la permitida por el protocolo HTTP), tal como su dirección IP, el puerto que se utiliza en la llamada, el método utilizado (GET, POST), etc.

- Permiten además la utilización de cookies y sesiones, de forma que se puede guardar información específica acerca de un usuario determinado, personalizando de esta forma la interacción cliente/servidor. Una clara aplicación es mantener la sesión con un cliente.

- Los servlets pueden actuar como enlace entre el cliente y una o varias bases de datos en arquitecturas cliente-servidor.

- Permiten la generación dinámica de código HTML.

b) Funcionamiento de los servlets.
La forma de funcionamiento de los servlets se puede resumir con los siguientes puntos:

- Leer cualquier dato enviado por el usuario: los datos normalmente se introducen por medio de la página Web, pero también pueden obtenerse a partir de un applet Java.

- Obtener otra información sobre la petición que se encuentra embebida en la propia petición HTTP: esta información se refiere por ejemplo a los cookies, el nombre del host de donde proviene la petición, etc.

- Generar los resultados: esta parte puede requerir acceder a una base de datos, invocar a una aplicación ... o simplemente computar los datos de entrada.

- Generar un documento con los resultados: debemos establecer el tipo de documento que va a ser devuelto (una página HTML, una imagen, un archivo comprimido, etc.).

- Establecer los parámetros apropiados para la respuesta.

- Enviar la respuesta al cliente: una vez que tenemos el formato del documento que entregaremos como respuesta y tenemos establecidos los parámetros de la comunicación enviamos la respuesta al cliente.

2. Servlets y JSP.

   a) Servlets VS CGI
   Como hemos comentado anteriormente los primeros servidores HTTP no incluían ningún mecanismo para generar respuestas dinámicamente, de aquí surgió la necesidad de desarrollar interfaces que permitieran comunicar el servidor con programas externos capaces de implementar dicha funcionalidad.
   La especificación CGI describe una interfaz estándar que sirve para que un servidor Web envíe solicitudes del navegador al programa CGI, y para que el programa CGI devuelva datos de respuesta al navegador a través del servidor Web.
   Los servlets ofrecen la misma funcionalidad que los CGI pero evitan ciertos problemas.
   A continuación citaremos algunas de las ventajas más significativas que ofrecen los servlets frente a las CGI:

   - Con CGI cada vez se realiza una petición se creaba un nuevo proceso para atenderla, esto implica una serie de consecuencias como tiempo para el cambio de proceso, memoria ocupada ,... Sin embargo, con los servlets no se genera un nuevo proceso, sino un nuevo hilo de ejecución.

   - Si tenemos varias peticiones simultáneas y estamos utilizando CGI tendremos en memoria tantas instancias del programa como peticiones, esto implica un gasto enorme de memoria si el número de instancias es muy grande y por lo tanto una degradación continua del rendimiento. Los servlets, al tener hilos de ejecución, solo necesitan una instancia del programa con lo que se consigue ahorrar muchos recursos.

   - Cuando un programa CGI termina de responder a una petición, el programa finaliza. Esto hace que cada vez que se realiza una nueva petición todo el programa tiene que ser cargado

de nuevo, establecer de nuevo las conexiones con las bases de datos,... Los servlets, sin embargo, permanecen en memoria entre peticiones, lo que hace que se gane mucho tiempo en las sucesivas peticiones porque el programa ya está cargado en memoria.

- Los servlets son independientes de la plataforma, ya que están escritos en Java y por tanto siguen el estándar API y están escritos para ser ejecutados en una cantidad enorme de servidores (Apache, Microsoft IIS, Java Web Server, etc.).

- Una vez que tienes un servidor Web añadir un servlet supone un ligero incremento en el coste, a diferencia de los programas CGI, que necesitan una gran inversión para obtenerlos.

b) Ventajas de JSP.
JSP tiene unas cuantas ventajas sobre muchas de sus alternativas. Aquí van unas cuantas de ellas:

- Frente a HTML estático: el HTML normal no puede contener información dinámica, así que las páginas HTML no pueden estar basadas en la entrada del usuario o en fuentes de datos del lado del servidor. JSP es tan fácil y cómodo que es bastante razonable aumentar las páginas HTML, que sólo se benefician ligeramente por la inserción de datos dinámicos. Una de las opciones más extendidas es utilizar la combinación Apache-Tomcat. El servidor Apache nos proporciona páginas HTML de forma eficiente y rápida, mientras el servidor Tomcat proporciona el contenedor necesario para los servlets y las páginas JSP. Para utilizar esta combinación también es necesario un "puente"que conecte ambos servidores, de tal forma que cuando se pide una página HTML sea Apache directamente quien la sirva, mientras que si la petición es de un servlet o una JSP Apache recibirá esa petición y la transmitirá a Tomcat, que será el encargado de servir esa petición.

c) Conclusiones.
Las tecnologías servlet y JSP de Java, no se plantean como dos alternativas a poder utilizar separadamente, sino como técnicas complementarias. Es más, las páginas JSP cuando se compilan se

transforman en servlets. Una de las ventajas importantes de usar cualquiera de estas dos técnicas es que se utiliza código escrito en Java, con todos los beneficios que ello conlleva: no hay necesidad de aprender un nuevo lenguaje, portabilidad, reutilización, etc. El problema de utilizar servlets directamente es que, aunque son muy eficientes, son muy tediosos de programar puesto que hay que generar la salida en código HTML con gran cantidad de funciones println. Este problema se resuelve fácilmente utilizando JSP, puesto que aprovecha la eficiencia del código Java, para generar el contenido dinámico, y la lógica de presentación se realiza con HTML normal. Sin embargo, cuando en una página JSP se necesita introducir mucha funcionalidad, es decir, introducir mucho código Java para generar el contenido dinámico de nuestra página, ese mismo hecho lleva a que el código de la página JSP no sea demasiado claro. Por lo tanto, el dilema está en decidir cuándo utilizar servlets y cuándo JSP. Lo ideal sería usar JSP cuando el dinamismo que se pretende no supone introducir mucho código Java en las páginas, puesto que esto oscurecería el código. Sin embargo cuando hay mucha funcionalidad y necesitamos mucho código Java, lo ideal sería utilizar una página JSP que llamase a un servlet que contenga la funcionalidad necesaria para que éste realice el trabajo y genere la respuesta, ocupándose el código JSP de presentar la información que devuelve el servlet.

Pillow.

1. La librería PiLLoW.

   - CIAO dispone de las siguientes librerías entre otras:

     - PiLLoW: Interfaz WWW/HTML.
     - Prolog shell.
     - Distribuición: Pizarras, concurrencia, agentes, ...
     - PLAI: Analisis global.

- OPC: Optimización global.

- Programación Web:
  - Aplicaciones CGI.
  - Ver documentos HTML estructurados como términos de Prolog.
  - Generar formularios HTML.
    - Ofrece estructuras que permiten representar varios elementos relacionados con los formularios HTML. Dispone también de predicados que realizan y simplifican la tarea de escribir un programa Prolog que actúe como aplicación CGI, es decir que se facilita la traducción de datos de formularios HTML a un diccionario.
    - Acceso a documentos WWW. Permite el acceso a documentos WWW, facilitando el desarrollo de aplicaciones tales como buscadores o analizadores de contenido.
  - Plantillas HTML/XML.
    - Los plantillas son ficheros que contienen código HTML/XML estándar, pero en los cuales se pueden poner huecos identificados a través de una etiqueta especial. Usando plantillas podemos definir interfaces gráficas fácilmente reconfigurables.
  - Escribir manejadores de formularios.
  - Acceder a documentos WWW.

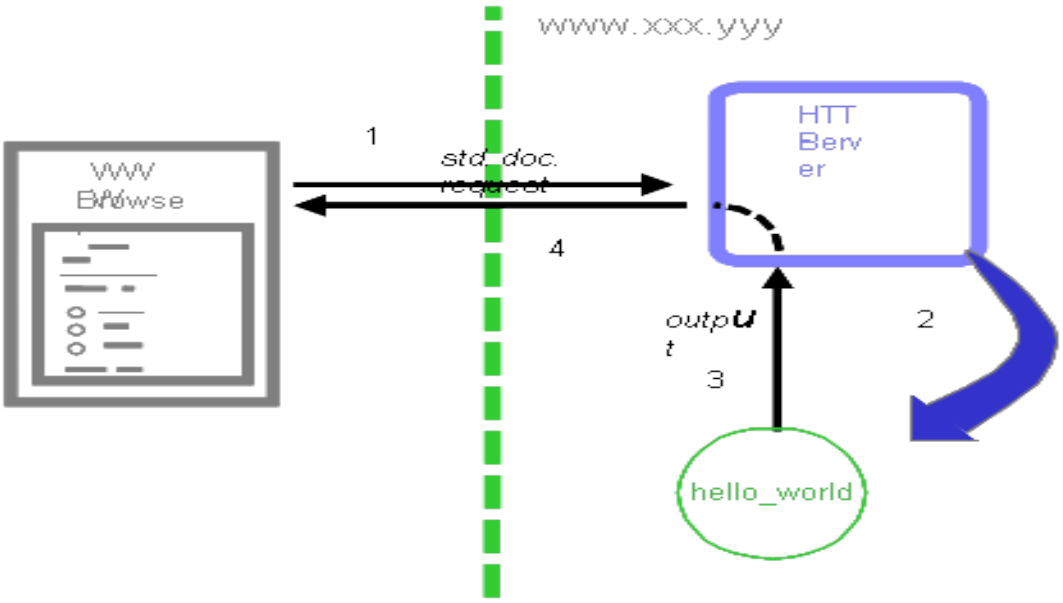
- Programación distribuida:
  - Pizarras.

- Diseño para un lenguaje LP/CLP distribuido.
- Modelado y acceso a servidores de información.
- Descarga automática de código para ejecución en local.

Para el desarrollo de la interfaz Web hemos utlizado PiLLoW que es, como hemos dicho anteriormente, una librería Prolog de dominio público diseñada para simplificar la Programación de aplicacionesWWW e Internet en sistemas LP/CLP.
Un aspecto importante de PiLLoW, que no hemos comentado antes es que permite manejar código HTML y XML como términos. Por lo tanto, tales estructuras sólo necesitan ser traducidas por los predicados apropiados a código HTML/XML para mostrar la salida. En general, esta relación entre el código HTML/XML y los términos Prolog permite ver una página HTML o un documento XML como un término Herbrand.

2. Pillow y las Aplicaciones CGI.
   Pillow hace uso de cgi para el desarrollo Web, la forma de funcionamiento de una cgi en prolog es la siguiente:

   - Se solicita la dirección de una aplicación CGI en el browser, p.e.: [12]
   - El servidor HTTP arranca el ejecutable 'hello_world'.
   - El servidor HTTP toma la salida del ejecutable (HTML/MIME).
   - Se le pasa al browser, que muestra el resultado.



- En Prolog:

```
main(_) :-
          write('Content-type: text/html'),
          nl, nl,
          write('<HTML>'),
          write('Hello <B>world</B>.'),
          write('</HTML>').
```

- Lo podemos compilar:

```
?- save('hello_word'), main(_), halt.
```

Con esta forma de actuar el problema surge con los ejecutables de gran tamaño.

Como dijimos antes HTML es estructurado y se puede reflejar esta estructura como términos Prolog. Facilitando la generación de la pagina de salida , entre otras cosas.

- Puede verse cualquier página Web como términos de Herbrand y manipularse fácilmente.

- PiLLoW provee dos predicados:

  a) output_html(F): Envía a la salida estándar el texto correspondiente al termino HTML F.

  b) html2tems(ASCII, Terms): Relates una lista de términos HTML y una lista de caracteres ASCII (reversible).

```
main(_) :-
        T = [ cgi_reply,
        html( [ 'Hello',
         b('world'),
        '.' ] ) ],
        output_html(T).
```

6.2.3. Desarrollo de la aplicación.

Como hemos comentado anteriormente, las cgi presentan limitaciones frente a los servlets y hoy en día ya no se usan tanto para el desarrollo de aplicaciones Web. El hecho de que hayamos utlizados CGI´s en nuestro proyecto se debe a que teníamos que integrar 2 tecnologías, servlets y jsp con otro entorno que es prolog que supone otro paradigma de programación con soporte específico para desarrollar análisis de código. Además, en el entorno en el que nos encontramos CIAO, se hace uso de la librería Pillow que utiliza CGI´s.

La arquitectura que presenta nuestra aplicación es como se muestra en la figura 6.1.

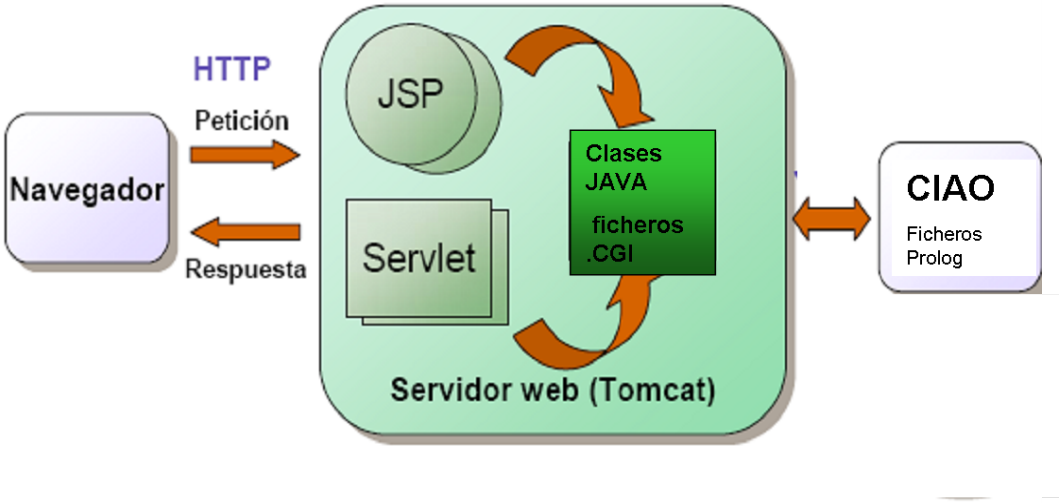


Figura 6.1: Arquitectura de nuestra aplicación

En el navegador, desde la correspondiente Web elaborada con JSP, el usuario enviará los datos que serán utilizados para dar respuesta a la petición del cliente. El servlet con la información de la jsp, sabrá lo que tiene que hacer con los datos y redirigirá el control para que se ejecuten los predicados o clases java para generar la respuesta a la petición del cliente que finalmente mostrará la información resultante de todo el proceso a través de una Web.

El árbol de directorios que presenta nuestra aplicación Web se presenta en la tabla 6.1.

```
Webapps/Checker
          Ficheros.html
          .jsp´s
          Imágenes
          RecursosEnGeneral
          WEB-INF
               classes:- *.class
               lib:- *.jar /*.zip
               cgi:- /*.cgi + recursos a los que accede la cgi
               web.xml
```

Cuadro 6.1: Árbol de directorios de nuestra aplicación Web

- En el mismo directorio de nuestra aplicación se encuentran las páginas web ya sean archivos .html o .jsp.

- Las imágenes que utilicemos se deben guardar en una carpeta de imágenes.

- Dentro de WEB-INF tenemos la parte correspondiente a los servlets, cuya implementación se encuentra en el subdirectorio clases.

- En el subdirectorio cgi además de las cgi propiamente dichas, tendremos los recursos a los que accede la cgi, lo que necesita para generar respuesta. En nuestro caso al tener que trabajar sobre un sistema en proceso de desarrollo por CLIP, nuestros recursos son toda la implementación ya realizada. Dado que esta información no la podemos almacenar en el subdirectorio cgi, hemos creados enlaces simbólicos desde la carpeta cgi a la carpeta en la que está el código que tenemos que usar para la correcta ejecución de nuestras cgi.

Durante el proceso de desarrollo nos hemos dado cuenta de que Pillow no es una de las mejores herramientas para elaborar aplicaciones Web, ya que como hemos visto han surgido muchos avances en este campo. Sin embargo, también es cierto que Pillow proporciona una buena forma de integración de código Prolog. Así pues, nuestro proyecto utiliza jsp y servlets para procesar elementos de información y para realizar tareas más tediosas que no tienen conexión con Prolog, como por ejemplo la página principal, que se encarga

de cargar un fichero con extensión .java y compilarlo para generar el fichero .class correspondiente y crear un archivo .pl con información necesaria para ser utilizada posteriormente. Además gracias a que usamos jsp esta página no tendrá que ser recompilada cada vez que cambiemos algo del código que ejecuta. Para la integración con Prolog, sí hemos utilizado Pillow. Es importante decir que el shell utilizado para los ejecutables no es el mismo que el shell interactivo que utiliza Ciao para ejecutar, compilar, ... los archivos prolog. Es decir en la cabecera de nuestras cgi tendremos algo de la forma:

```
#!/bin/sh
exec  /Systems/CiaoDE/bin/ciao-shell 0*
:- use_package(pillow).
:- use_package(default).
```

... seguiriamos importando los módulos que vayamos a usar y posteriormente tendríamos el main.
Por ejemplo:

```
main(_) :- get_form_input(Info),
           get_form_value(Info,complexity,Complexity),
```

Aquí lo que estaríamos haciendo con get_form_input(Info) es coger el Diccionario de la página Web (la información que nos llega del formulario). Tendríamos pares compuestos por el nombre del elemento Web (complexity) y su correspondiente valor introducido por el usuario en el formulario Web. Para tomar el valor que ha sido introducido en el elemento de la página Web llamado complexity usamos get_form_value (Info,complexity,Complexity), de esta forma en Complexity ya tendríamos el valor que ha sido introducido en el formulario y con el que podríamos llamar a los predicados Prolog para generar la respuesta a la petición que hizo el usuario.

¿Qué ocurre tras la interfaz?

Al comienzo de la aplicación se muestra una interfaz para que el usuario pueda cargar un programa implementado en java. A partir de ahí, lo que el sistema hará será compilar el .java para generar el .class correspondiente.

Ahora ya tenemos el bytecode del archivo cargado, a partir del cual trabajaremos.
El primer paso, oculto tras la interfaz consiste en hacer uso de un predicado (load_classes) implementado en Prolog gracias al cual conseguimos cargar la clase en el sistema y generar hechos del tipo class y method, con estos hechos podemos extraer la información necesaria acerca del nombre de la clase, de los métodos de la clase, las variables que utiliza... Para poder hacer el checker de uno de los métodos será necesario ofrecer al usuario la posibilidad de elegir el método sobre el cual quiere comprobar su complejidad y generar sus ecuaciones de coste simplificadas.
Una vez que ya tengamos el método seleccionado, la clase cargada y los respectivos hechos class y method, el paso siguiente a da consiste en crear un archivo que contenga dos predicados prolog llamados benchmark e infofix_point, elaborados a partir de los datos que obtenemos de los class y method. En benchmark tendremos el nombre de la clase, la lista de clases, structure_loops a false y eliminate_stack_vars a true. Infofix_point será un predicado que tiene más que ver con el método sobre el cual se va a checkear su complejidad, así pues estará formado por el nombre de la clase tal y cual no los da el hecho method y una lista de elementos cuyo tamaño es igual al número de variables del método entre otras cosas.
Benchmarks e info_fixpoint presentan el siguiente aspecto:

```
benchmark('Hanoi',['Hanoi'],[structure_loops(false)],
[eliminate_stack_vars(true)]).

info_fixpoint('Hanoi',
entry(['Hanoi:hanoi(IIII)V',[A,_B,_C,_D],[]]),
domain(size_relations)).
```

La información para este último predicado es obtenida del hecho method que nos da los siguentes valores una vez cargada la clase Hanoi, por ejemplo:

```
A = 'hanoi(IIII)V',          ⇒ Es el nombre del método
tal cual debe ir en info_fixpoint.

B = 'Hanoi',          ⇒ Nombre de la clase.

C = hanoi,          ⇒ Nombre del método original.

D = [primitiveType(int),primitiveType(int),primitiveType(int),
primitiveType(int)],          ⇒ Argumentos
```

...

Una vez que ya tengamos toda esta información ya estamos en disposición de pasar a la siguiente fase. Nos situamos ahora en una segunda interfaz. Aquí el usuario sólo tendrá que terminar el proceso de petición de datos para llevar a cabo la comprobación de la complejidad. Se le pedirá que elija el orden que quiere chequear linear, polinómico, exponencial, logarítmico y el grado del orden seleccionado. Con estos datos y con todo lo generado en el primer paso, ya estamos en disposición de utilizar el predicado Prolog checkingComplexity que afirmará o no si el orden introducido se corresponde o no con el orden real del método y además podremos ver las ecuaciones de coste del método.

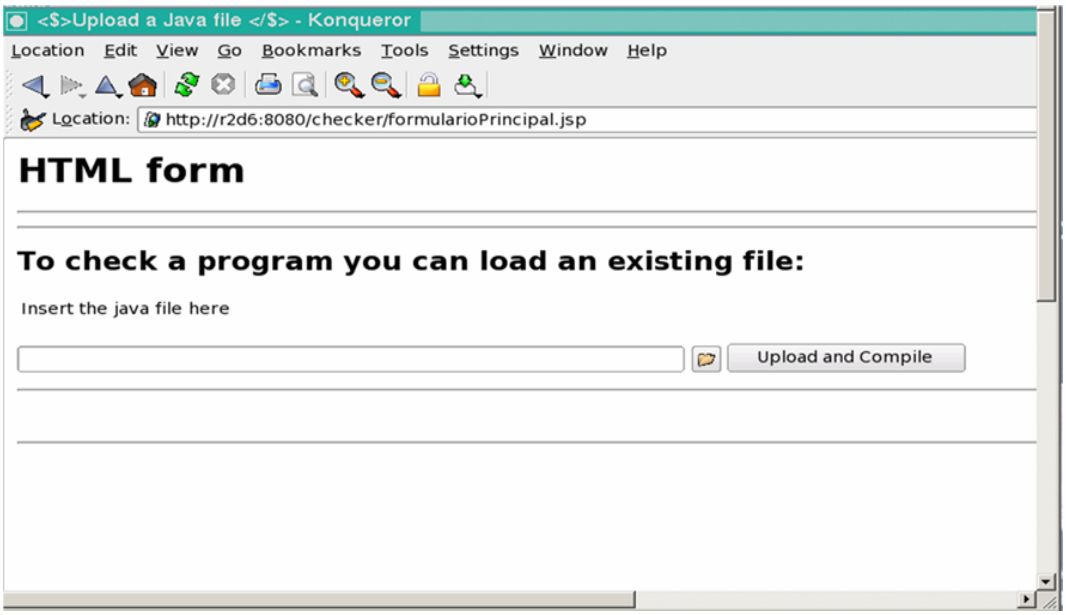
Si todo ha funcionado bien, después de todo este proceso el usuario podar disponer de la información que solicitó.

Conclusiones de la integración.

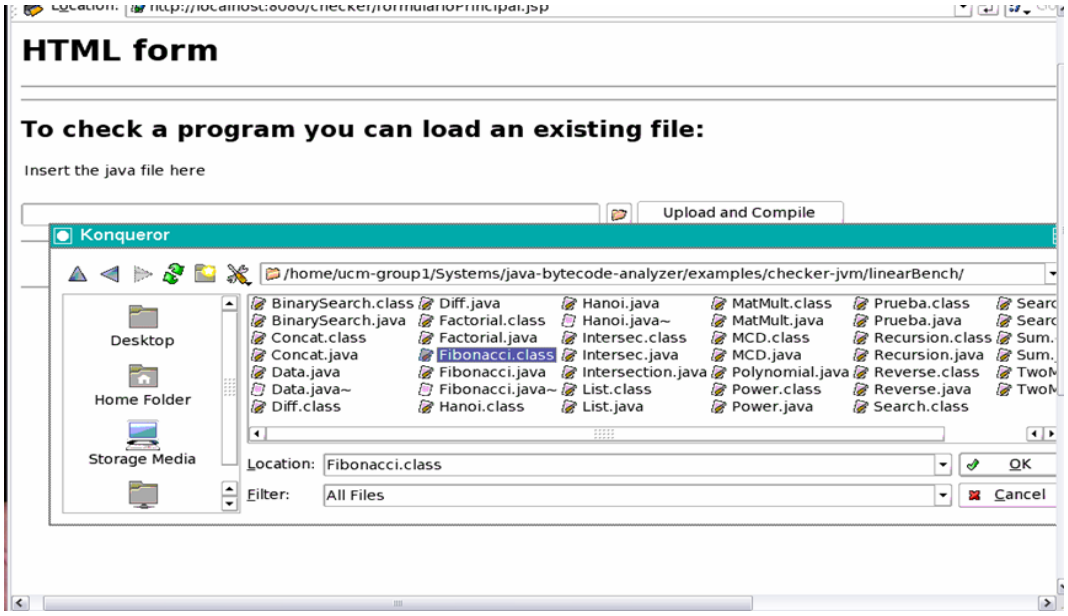
La integración de esta aplicación ha sido una dura tarea a realizar, ya que nos encontramos en un proyecto que ya estaba en desarrollo, por eso hemos tenido dificultades que nos ha llevado tiempo solventar ralentizando el proceso de implementación de la aplicación Web. También surgieron problemas con la utilización de Pillow y con la puesta en marcha de la aplicación en el sistema. Las jsp y servlets nos simplificaron, en parte, el proceso de construcción, y Pillow nos proporcionó un soporte a la hora de integrar el código Prolog.

6.3. Uso del sistema

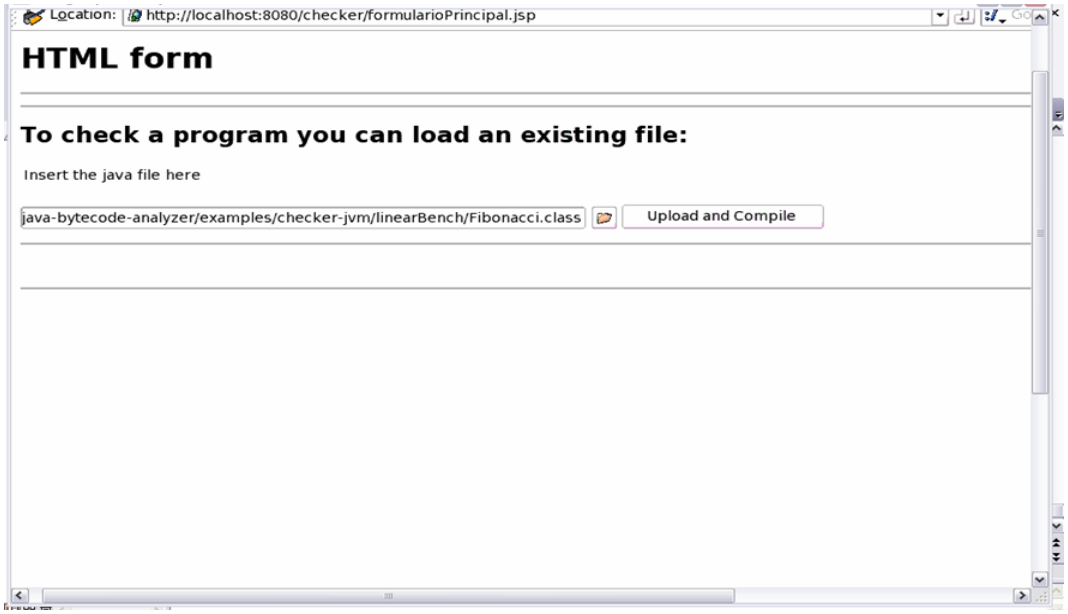
6.3.1. Archivo cargado y compilado con éxito



Desde esta página tenemos que cargar el programa Java del que deseamos comprobar la complejidad de alguno de sus métodos.
Para cargar el programa sólo tendremos que pulsar sobre el icono marcado con una carpeta, y se nos abrirá una ventana de exploración para localizar el archivo que queremos cargar.

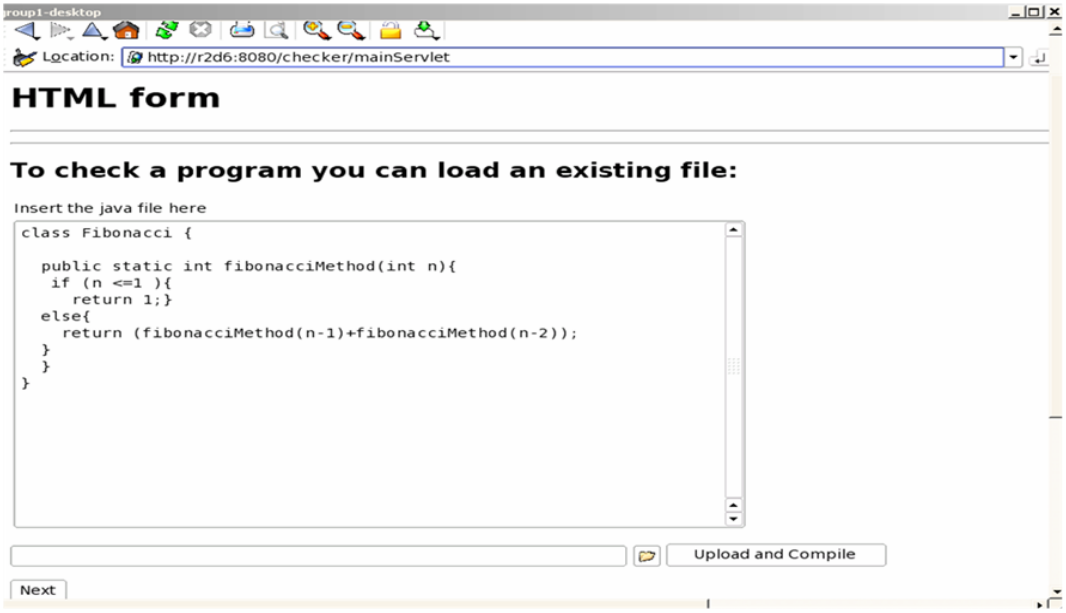


Una vez seleccionado el archivo le damos a ok y aparecerá la pantalla inicial, pero ahora tendrá el nombre del archivo que hemos cargado.



Para llevar a cabo la compilación del código cargado tendremos que pulsar

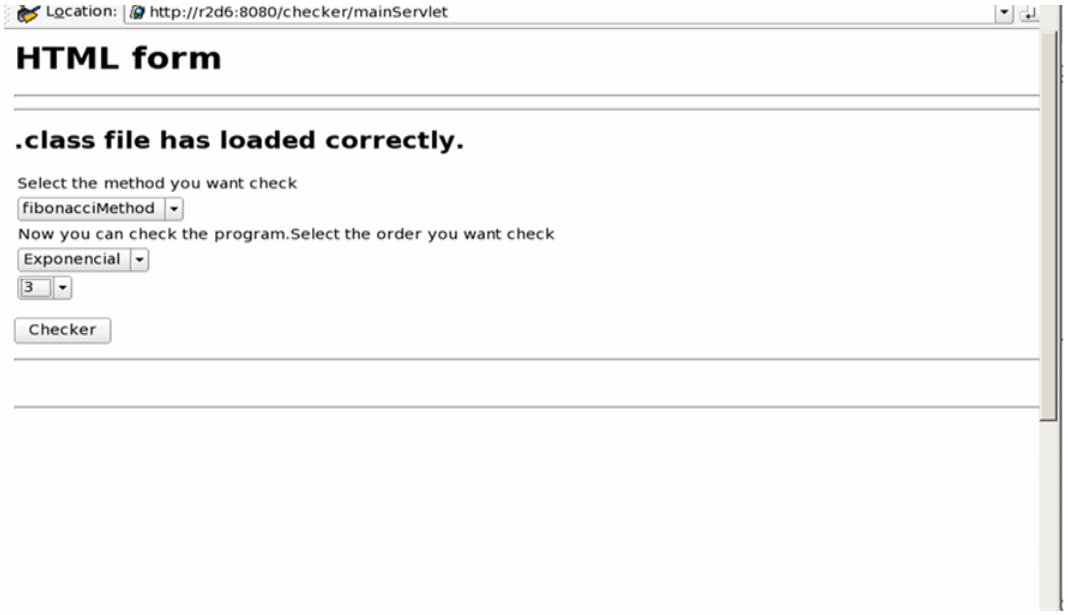
sobre la tecla 'Upload and Compile'. El resultado de tal acción es la siguiente:



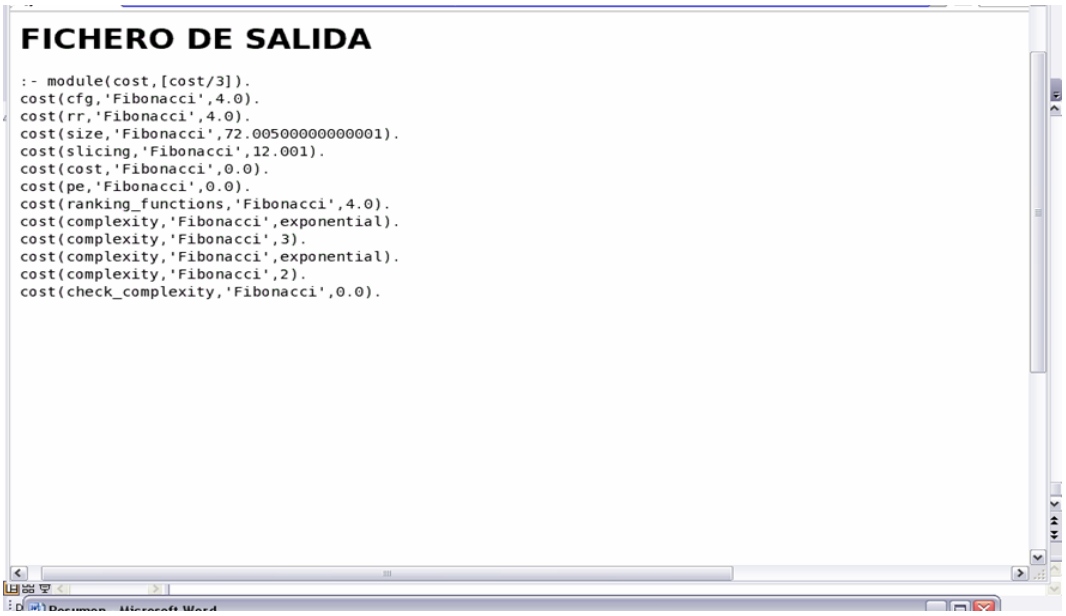
Vemos que el programa que hemos cargado es el que queríamos y que no tiene errores de compilación. Lo siguiente que tenemos que hacer es pulsar el botón 'Next'.

Si hemos llegado a esta página, quiere decir que el programa Java se ha compilado correctamente y tenemos acceso a la interfaz que nos permite llevar a cabo la comprobación de complejidades para nuestro programa.

Lo que tenemos que hacer ahora, es seleccionar el método que queremos checkear, seleccionar la complejidad que queremos comprobar y el orden. (En el ejemplo la complejidad elegida es exponencial de base 3).

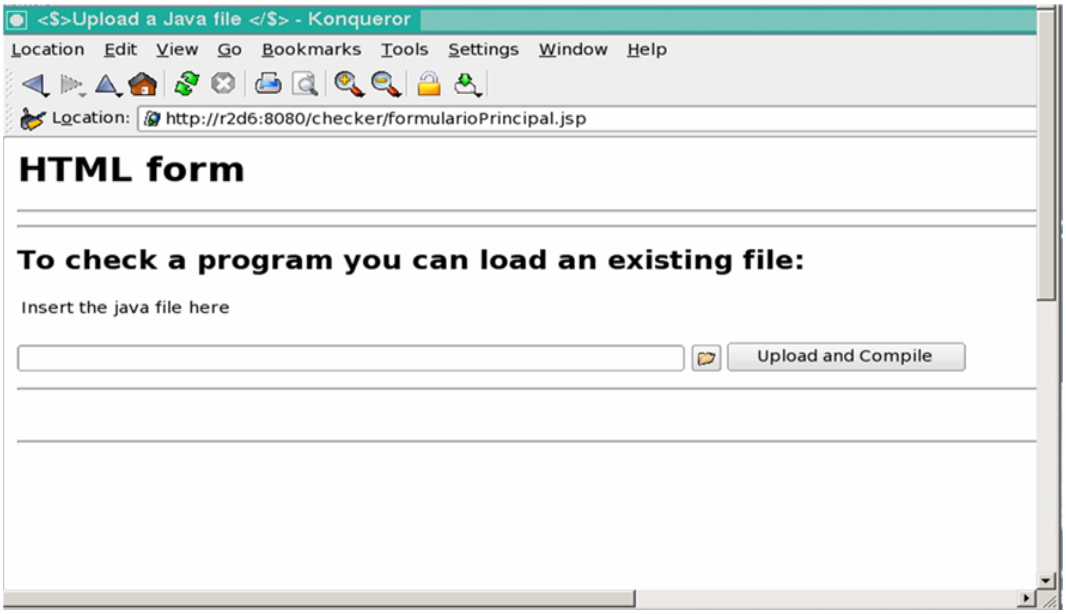


Una vez que tengamos elegidos todos los parámetros con los que queremos comprobar la complejidad, sólo tendremos que pulsar sobre el botón çheckerz obtendremos como resultado un archivo con los resultados experimentales, entre los que se encuentra la información referente a la complejidad. (En el ejemplo podemos ver que es exponencial de base 3 pero afina diciendo que es de base 2).

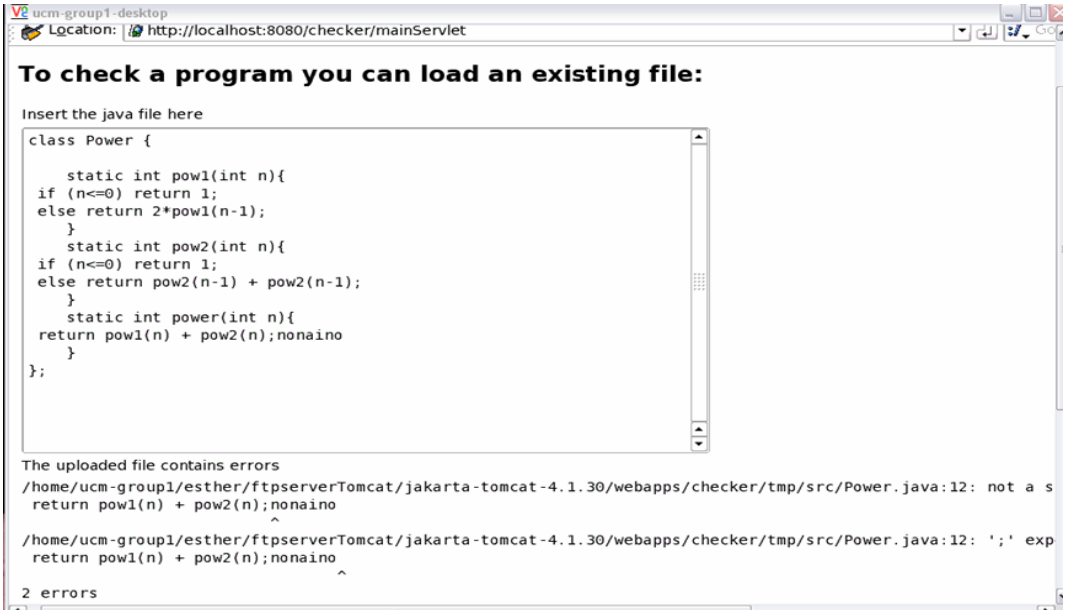


6.3.2. Archivo compilado sin éxito

Procedemos igual que siempre cargando el archivo JAVA pulsando sobre el icono marcado con una carpeta.



Sin embargo, esta vez cuando se da al botón de Üpload and Compile"lo que ocurre es que se muestra el archivo pero además se muestran los errores surgidos en el proceso de compilación. De esta forma hasta que el usuario no corrija tales errores no podrá realizar el checkeo de la complejidad de ninguno de los métodos.



Una vez solventados los errores se procederá igual que en el caso en el que se carga y compila correctamente.

Bibliografía

[1] J. D. Ullman. A. V. Aho, J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] R. Sethi A. V. Aho and J. D. Ullman. *Compilers - Principles, Techniques and Tools*. A - W, 1986.

[3] A. Coglio and A. Goldberg. *Type safety in the JVM: Some problems in the Java 2 SDK 1.2 and proposed solutions*. Concurrency and Computation: Practice and Experience 13, 2001.

[4] Felten E. W. Dean, D. and D. S. Wallach. Java security: From hotjava to netscape and beyond. *Procedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, 1996.

[5] R. D. Dean. *Formal aspects of mobile code security, Ph.D. thesis*. Princeton University, 1999.

[6] S. Genaim G. Puebla D. Zanardini E. Albert, P. Arenas. Automatic cost analysis of java bytecode. *Technical Report CLIP10/2006.0, Technical University of Madrid, School of Computer Science, UPM*, December 2006.

[7] S. Genaim G. Puebla D. Zanardini E. Albert, P. Arenas. Experiments in cost analysis of java bytecode. *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2007)*, Electronic Notes in Theoretical Computer Science, Elservier-North Holland, 16 pages, 2007.

[8] S. Genaim G. Puebla D. Zanardini E. Albert, P. Arenas. Cost analysis of java bytecode. *In the 16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science. Springer, March 2007. Available online `http://clip.dia.fi.upm.es/papers/jvm-cost-esop.pdf`.

[9] S. Genaim G. Puebla D. Zanardini E. Albert, P. Arenas. Computing upper bounds of cost relations using programmig languajes techniques. Trabajo en progreso.

[10] S. Genaim G. Puebla D. Zanardini E. Albert, P. Arenas. Transforming java bytecode to a structured representation for improving analysis accuracy. Trabajo en progreso.

[11] Bruce Eckel. *Piensa en Java*. Prentice Hall, Segunda edición.

[12] Disponible en. `http://www.xxx.yyy/cgi_bin/hello_world`. 2007.

[13] M. Carrao M. Hermenegildo P. López-García G. Puebla F. Bueno, D. Cabeza. The ciao system. reference manual(v1.13). *Technical report, School of Computer Science (UPM). Disponible en http://www.ciaohome.org*, 2006.

[14] S.Ñ. Freund and J. C. Mitchell. A type system for object initialization in the java bytecode language. *ACM Transactions on Programming Languajes and Systems*, 1999.

[15] SthephenÑ. Freund and John C. Mitchell. A type system for the java bytecode languaje and verifier. *Journal of Automated Reasoning*, 2003.

[16] Peter Haggar. Java bytecode: Understanding bytecode makes you a better programmer. *Disponible en* `http://www-128.ibm.com/developerworks/ibm/library/it-haggar_bytecode/`, 2006.

[17] M. Rosendhal. *Automatic Complexity Analysis*. In Proc. FPCA. ACM, 1989.

[18] McDirmid S. Sirer, E. G. and B. Bershad. *Kimera: a Java system architecture*. 1997.

[19] F. Spoto. *JULIA: A Generic Static Analyser for the Java Bytecode*. In FTfJP, 2005.

[20] F. Yellin T. Lindholm. *The java Virtual Machine Specification*. A-W, 1996.

[21] Mathematica: the Way the World Calculates. `http://www.wolfram.com/products/mathematica/index.html`.

[22] F. Tip. *A Survey of Program Slicing Techniques.* J. of Prog. Lang., 1995.

[23] A. Tozawa and M. Hagiya. Careful analysis of type spoofing. *Java-Informations-Tage*, 1999.

[24] H. S. Wilf. *Algorithms and Complexity.* A. K. Peters Ltd, 2002.